# Just In Time Compilation for a High-Level DSL

**Nathan Dunne**

**1604486**

**3rd Year Dissertation Project**

**Supervised by Gihan Mudalige**

Department of Computer Science

University of Warwick

2019–20

## Abstract

The OP2 Domain Specific Language was originally designed 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 portion of the OP2 Framework, 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 optimisations 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 conventionally 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 (GPUs) have become an increasingly popular hardware architecture, having originally been conceived as specialised 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 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 separate the application code from the optimisation requirements. OP2 is already able to generate optimised code for a number of back-ends from an application file.

This report details an investigation into applying a new optimisation to the GPU code generation in OP2, and the process of benchmarking the performance gain, if any, it is able to provide. This 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 Professor 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 existing codes.

Since OP2 is Open Source and freely available, the implementation I produce would become part of the framework, 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 predicting weather patterns. It is usually necessary to discretise such problems, which means dividing a continuous space into a number of discrete cells.

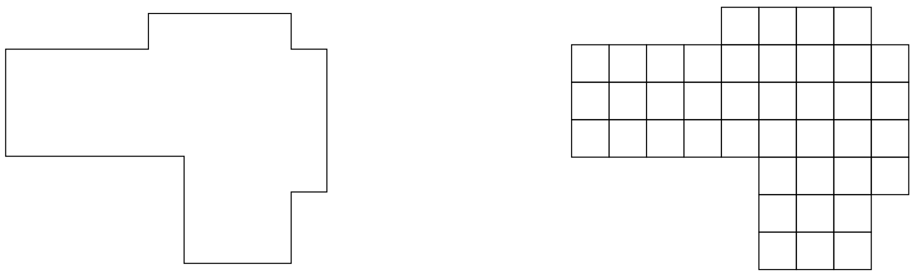


Figure 1: Example 2D decomposition

Depending on its structure, a mesh can be described as either structured (regular) or unstructured. Structured meshes such as Figure 2 are made up of cells following a regular pattern, while unstructured meshes use connectivity information to specify the mesh topology, as Figure 3. OP2 is designed for unstructured mesh solvers, but since a structured mesh can be represented using connectivity information, it is a "stronger" category that includes structured meshes.

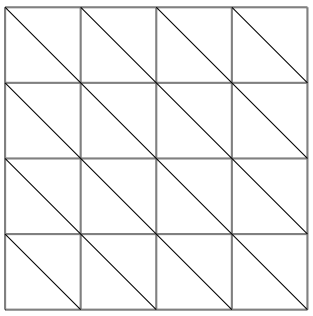


Figure 2: Tri-Structured Mesh

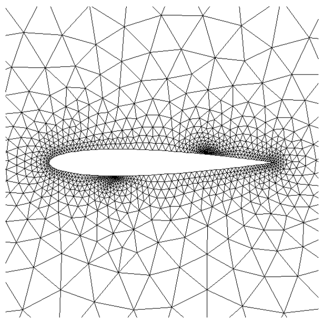


Figure 3: Airfoil Tri-Unstructured Mesh

A particular simulation might, for example, be approximating the velocity of a fluid in each cell, and at every time step re-calculate this value based on the values of cells around it. Programs which make use of this repeating pattern to calculate value are commonly referred to as a stencil codes [36]. An example of a 2D stencil is shown in Figure 4.

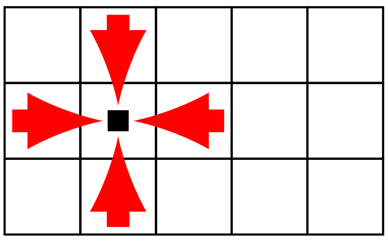


Figure 4: Example 2D stencil code

## 1.3 Report Structure

The rest of this report is structured as follows: In Section 2 (p5) the research done to inform the work in this project is discussed, along with relevant similar academic work. This is followed by a Project Specification in Section 3 (p16), which includes the requirements of the project, and a high level plan of the implementation to be completed, its components, and how they will interact. Section 4 (p24) details the Implementation itself, and the expected results of code generation with JIT compilation enabled, and Section 5 (p51) explains the plan for, and outcome of, Testing and Benchmarking using an example application on a HPC cluster.

Section 6 (p62) contains an Evaluation of the work completed, including a discussion of Future Work (p62), and a reflection on Project Management (p65) which could build on top of what was done for this report, and lastly Section 7 (p69) provides an overall Conclusion

# 2 Research

## 2.1 NVidia CUDA Programming Model

Since this project will require the automatic generation of GPU code source files from sequential code, it is important to introduce the NVidia hardware, and the C Application Programming Interface (API) which will be utilised. NVidia's Compute Unified Device Architecture (CUDA) is used to program their GPUs, which is a proprietary language. Relevant information for this project from the *NVidia CUDA C Programming Guide* [7] has been summarised below, and it will continue to be used for reference throughout this report.

### 2.1.1 Hardware

The GPU is a computer component that exists alongside the CPU, and communicates on the Peripheral Component Interconnect express (PCIe) bus. The CPU is able to pass workloads over to the GPU for it to execute, particularly compute-heavy workloads which would bottleneck the CPU. As can be seen in Figure 5, the GPU has its own onboard memory, and cannot access the computer's RAM directly, so any data that the GPU will process must be copied across the PCIe bus and stored in the GPU's local memory, at the expense of time.

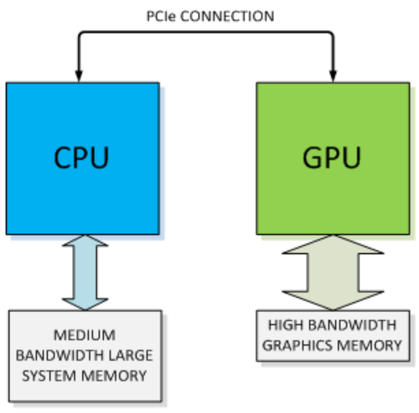

PCIe CONNECTION
CPU
GPU
MEDIUM BANDWIDTH LARGE SYSTEM MEMORY
HIGH BANDWIDTH GRAPHICS MEMORY

Figure 5: CPU - GPU communication. Diagram from [13]

Figure 6 contrasts the design of a traditional CPU to that of a GPU, where area of a component corresponds to resources devoted to it. Since the Arithmetic Logic Unit (ALU, coloured green) is responsible for arithmetic operations, and the Cache and DRAM components (coloured orange) will speed up memory operations, it should be clear why the GPU is ideal for compute-heavy workloads, where the ratio of arithmetic operations to memory operations is high.

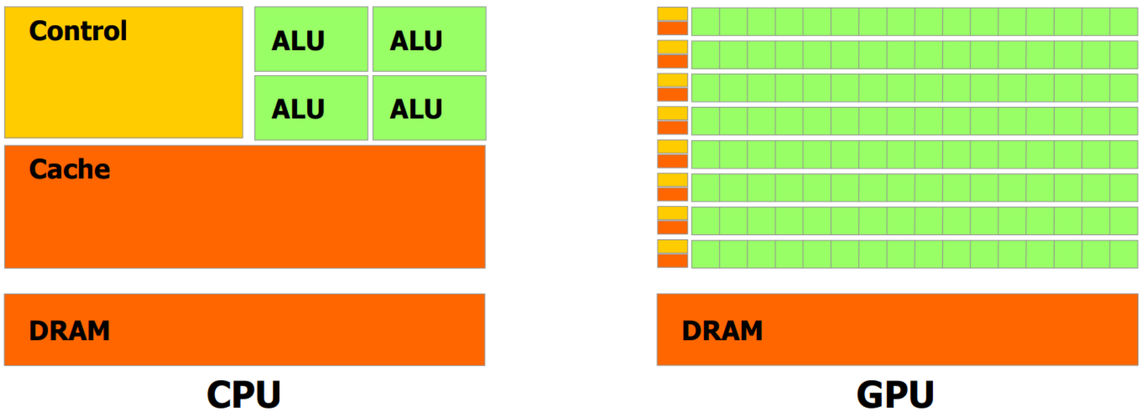


Figure 6: Architecture Comparison. Diagram from [7, p3]

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 often need to be repeatedly applied to all edges, cells, or nodes in a given mesh.

### 2.1.2 GPU Parallelism

Workloads executed on a GPU are divided among a Grid of Blocks, where each Block contains a number of Threads, all executing simultaneously in parallel. To allow for easier mapping from the problem space to a Thread Identifier, the Block ID and Thread Identifiers within each block can be 1D, 2D or 3D [7, p9]. Figure 7 shows an example where 2D identifiers are used.

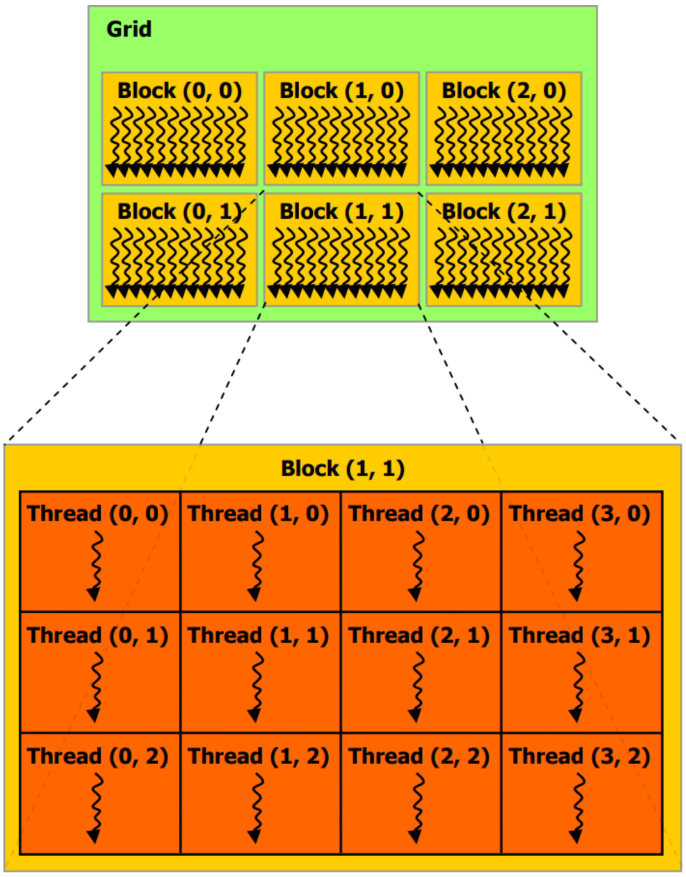


Figure 7: 2D grid of Blocks and Threads. Diagram from [7, p9]

### 2.1.3 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 [7, p15], and executed on an NVidia GPU device. The difference however, 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 that is already executing on the device [7, p81].

Global functions therefore act as a sort of entry point into device code. They are called using additional "execution configuration" syntax [7, p7], added as a C language extension by NVidia, which allows the user to specify special parameters for the required 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 a normal, C language `int` type (1D), or use a CUDA type `dim3` [7, p9] to specify up to 3 dimensions. Any value left unspecified is initialised as 1 [7, p87]. The function body will then be executed `num_blocks` $\times$ `threads_per_block` times, with all threads beginning at the same time. The number of threads has an upper limit depending on the hardware, for example the Kepler Architecture can support 2048 total threads per multiprocessor [10], for example 8 blocks of $16 \times 16$ threads (2 dimensions of thread IDs).

Inside the function body built in variables `threadIdx.x`, `blockIdx.x`, and `blockDim.x` can be used to determine the work which a certain thread should carry out, usually by calculating an array index from their values. Appendix A is a CUDA program written during research to build familiarity with writing CUDA code, which utilises these constructs and ideas.

In the next section, the OP2 Framework and its existing code generation will be discussed. It can produce optimised code executable on a GPU from unoptimised sequential code, and parts of this code generator will aid in development of the new code generation script being produced for this report, with the Just-In-Time Compilation functionality as an addition.

## 2.2 OP2

### 2.2.1 Existing Work

This project is focussed on a contribution made to the OP2 open source library. There is a large quantity of literature available on the OP2 website [32], but the following section aims to provide enough understanding for someone unfamiliar with OP2 to follow the rest of this report.

OP2 is an "active library" framework [18], which takes a single application code written using the OP2 Application Programming Interface (API), embedded in either C or Fortran and uses source-to-source translation to produce multiple different source codes, each targeting a different optimised parallel implementation. The generated source code is then compiled, and linked against the OP2 library files to produce an executable for the original application which will run on the desired platform. It is the extra step of code generation that makes OP2 an "active" library, compared to conventional software libraries.

Since this project is focused on the GPU back-end, the journal article *Designing OP2 for GPU architectures* [30] is necessary background material, as it covers a lot of important details from the implementation of the existing GPU framework.

**Designing OP2 for GPU architectures:** This article, originally published in the Journal of Parallel and Distributed Computing in 2013, describes the key 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), where an operation relies on another being complete before it can begin, otherwise the result may be incorrect. 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 which perform a read-modify-write operation as a single, uninterruptible action on a 32-bit or 64-bit word residing in global or shared memory [7, p96]. This means that a thread cannot alter the value in memory between the read and write operations of another thread, which could cause a data dependency to be violated, as all three operation are performed as a single atomic action, and therefore they cannot overlap.
In the implementation for this project, atomic operations were selected as the best solution for this issue.

The paper also introduces the consideration of data layout in memory. Figure 15 demonstrates the different layouts possible when there are multiple components for each element, in this case 4 elements with 4 components each. While using Array-of-Structs is the default layout, and the easier to implement, the paper concludes that transforming application code to utilise the Struct-of-Arrays layout is effective for reducing the total amount data transferred to and from GPU global memory, in some cases by over 50%.

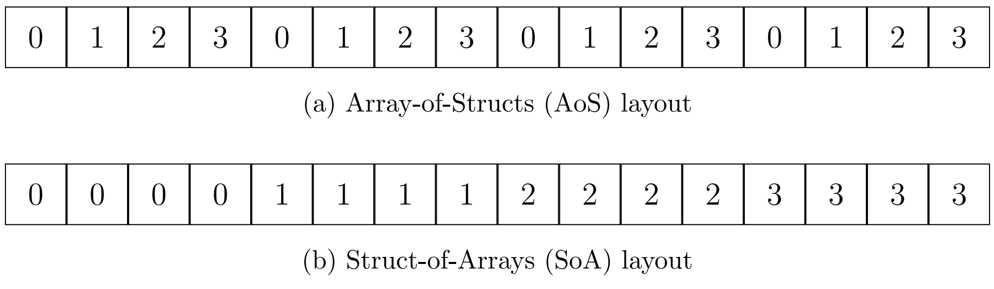

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
(a) Array-of-Structs (AoS) layout
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
(b) Struct-of-Arrays (SoA) layout

Figure 8: Data layouts. Diagram reproduced from [30]

The SoA layout is enabled by setting the value of an environment variable: `OP_AUTO_SoA=1` The environment must be set prior to code generation [17, p13], as it modifies the output of the code generation stage. In the Implementation section (Section 4) the differences in generated code when this is enabled will be explained in greater detail.

The existing solution is able to generate optimised CUDA code for a parallel loop where the resulting code can map set elements to a GPU thread processed by which it will be processed, therefore operating on many set elements at once in parallel. 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 bridge this gap, 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 [17, p4], the consideration for not violating data dependencies between iterations is removed in the generated code, and therefore 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 the OP2 active library framework, which would immediately benefit from further optimisation of the generated code, including: Airfoil [16] - a non-linear 2D inviscid airfoil code; Hydra [24] - Rolls Royce's turbo-machinery simulator; and Volna [22] - a finite-volume nonlinear shallow-water equation solver used to simulate tsunamis.

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

A further benefit is that such applications can be ported onto a new generation of hardware, which could be developed in the future. Only the OP2 back-end library would need to be modified to provide support the new hardware, instead of every application individually. This portability can save both time and money in development of HPC applications if multiple different hardware platforms are desired to be used.

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

### 2.2.3 OP2 Results

The optimisation of the Hydra turbo-machinery simulator was presented in a 2016 paper titled *Acceleration of a Full-scale Industrial CFD Application with OP2* [34]. This paper compares the newer OP2 framework with it's predecessor OPS [35], as well as benchmarking the application against OP2 code generated utilising OpenMP [38] and MPI [14] for thread and process level parallelism respectively. Initially, OP2's GPU code generation was outperformed by both OPS generating MPI, and most of the OP2 MPI implementations - completing 45% slower than the best CPU performance (2 CPUs, 24 MPI processes).

However, after some parameters had been tweaked including modifying the block sizes and enabling the Struct-Of-Arrays data layout, a single K20 GPU was able to achieve nearly 1.8× speedup over the original OPlus version, and close to 1.5× over the MPI version of Hydra with OP2. It is worth noting that these MPI implementations are already optimised parallel versions, not sequential implementations, so a speedup of 150% is a very significant result.

## 2.3 Just-In-Time Compilation

The term “Just-In-Time Compilation” is most commonly associated with the Java programming language, and particularly the Java Run-time Environment (JRE), as “JIT” Compilation is an integral part of the design and usage of the Java Virtual Machine (JVM).

The Java compiler (`javac`) compiles code into platform independent ”bytecode” [28], then at run-time this bytecode can be interpreted, or compiled a second time by the JVM into native code, and optimised specifically for the machine it is running on. It can also take the program’s inputs into account, since they will be known and fixed at run-time.

This re-compilation from bytecode to native code is only done for “hot” sections which are dominating the runtime [28], while the rest continues to be interpreted, as it is not worth the time cost to recompile. Chapter 4 of *Java Performance: The Definitive Guide* [31] contains further detail on the JIT compiler and its impact on the performance of Java.

The core idea of recompiling code at run-time to obtain performance is the inspiration for the new optimisation investigated in this report, and the origin of its name. However, the Java approach does not exactly map onto OP2. In the existing framework, there is no possibility of intermediate code, and no Virtual Machine in which the binary will execute that can profile the running code and react to “hot” sections. Instead, an alternative source file at the same “level” above native code is created (see Figure 9). This new source code that has been translated will be able to utilise assertions made using the input data, and so is compiled and used in place of the equivalent but unoptimised functions compiled ahead of time. The design of the JIT compilation system for OP2 will be covered in greater detail in Section 3.

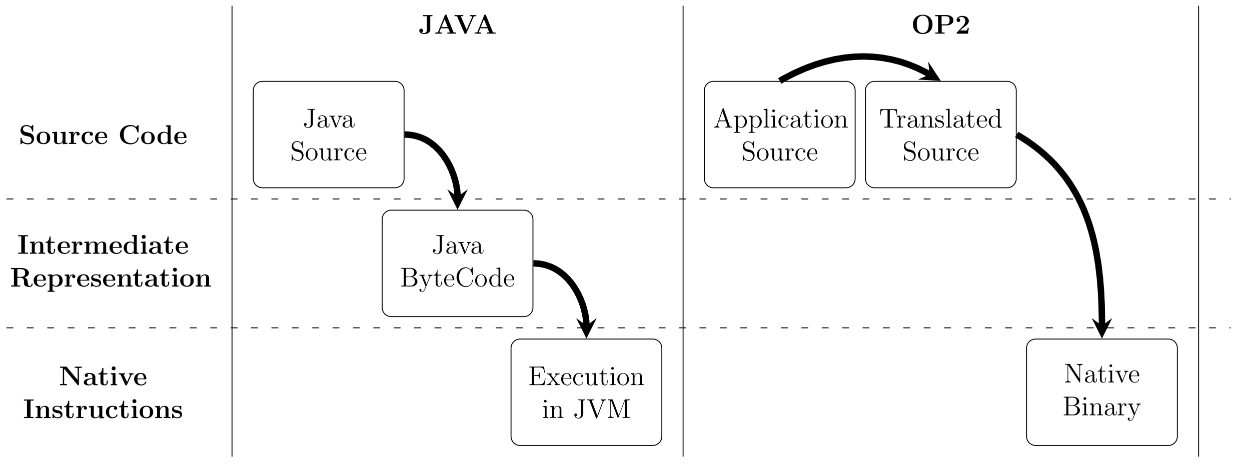


Figure 9: Comparison of Java and OP2 JIT

### 2.3.1 Related Work

While Java's JIT compilation gives a good indication that there is real benefit to be gained from using the technique, its implementation does not translate well on to the active library workflow. While researching more similar applications of the concept, the *easy::JIT* library was discovered.

**easy::JIT:** *easy::JIT* [29] is a library created by Juan Manuel Martinez Caamaño and Serge Guelton of *Quarkslab* [1]. It targets C++ code, and utilises `clang` [2] as the compiler, which is built using the Low Level Virtual Machine (LLVM) framework. It 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 generate all 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 like `clang`, although a translator using LLVM Intermediate Representation to replace the current Python and MatLab source-to-source translation scripts is currently in development [15]. This would bring it more in line with Java, having an original source, an intermediate representation, and then native code after the second compilation stage.

The implementation completed for this report is building on the compiler agnostic OP2 implementation, and therefore will not utilise LLVM .

# 3 Specification

In order to clearly explain the design of the extension that will be made to the OP2 framework, it is important to first understand the existing work-flow. The following is a high level overview of the components of OP2 that pre-date this project, so that when the new system is described in Section 3.2 it is easier to understand.

## 3.1 Existing System

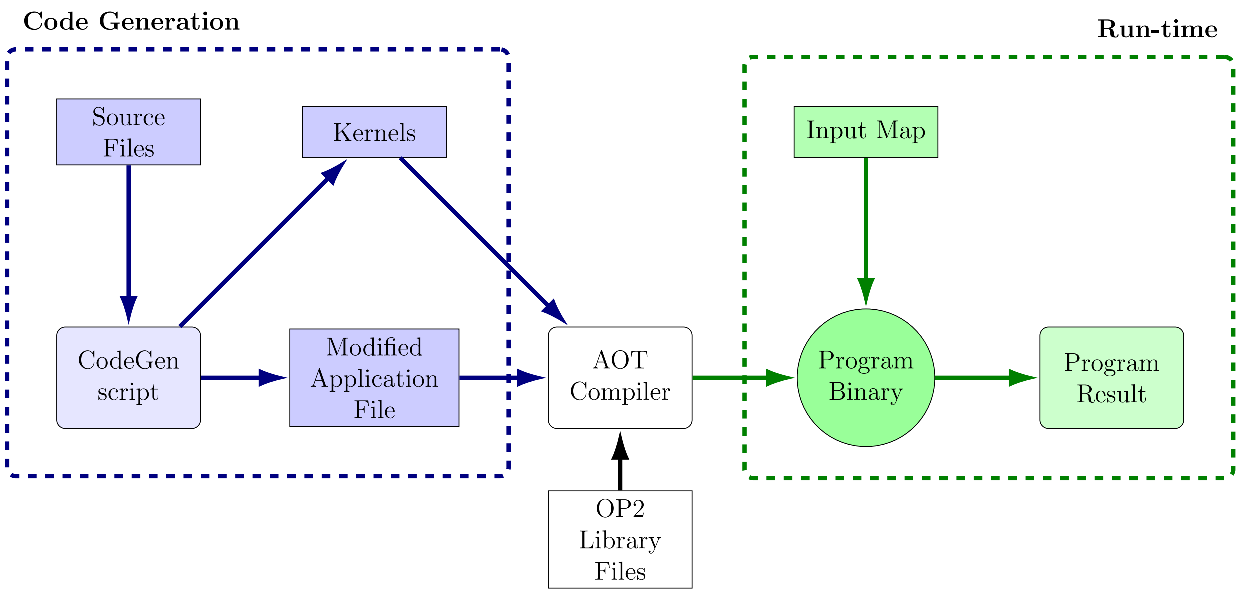


Figure 10: Existing OP2 System Diagram

The pre-existing OP2 workflow is shown in Figure 10. The diagram starts in the top left with the Code Generation stage, beginning at the system's input: a set of Source Files. The set of Source Files cannot be empty, and must contain at least one file, called the Master Application file. The Master Application file is a normal C program, containing OP2 API calls to define of sets, maps, and constants, as well as initialising and cleaning up the OP2 execution with the `op_init()` and `op_exit()` Library functions. It describes the structure of the application.

The input Source Files can optionally contain addition C source and header files, included in the normal way with `#include` statements, to assist with the organisation of a complex application with a large code-base. An example usage of these files would be a header file for each parallel loop, containing the function to be executed as the body of the loop.

These Source files are parsed by the code generation Python script, from which the output is: a modified version of the master application file, and a kernel file for each parallel loop. The output files are compiled, and linked against the OP2 library files 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, as they are only known at runtime, after the compiler has completed and the binary is being executed. This is where the implementation for this project comes in: to provide the ability to use the input data when optimising.

### 3.1.1 op2.py

The code generation is done using Python scripts, with the main script being `op2.py`, which parses the input files to gather data, and provides this data to a number of other scripts, which each perform the code generation for a specific hardware platform. It uses the Python Regular Expressions (RegEx) library: `re` [33] to identify OP2 API calls in the Application File, and ensures certain conditions are met - for example that `op_init` and `op_exit` are both called at least once to initialise and clean up the OP2 execution environment.

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

Once the Application has been analysed, `op2.py` produces a modified copy of the Application File, named `[application]_op.cpp`, which is largely the same as the file provided by the application programmer, but with the addition of `extern` declarations for the function each parallel loop will call: `op_par_loop_[name]`. Defining a function `extern` means it has external linkage, and therefore the definition of the function may be found during the linking stage of compilation, not in the current pass.

The generator scripts for each platform will receive the list of loop details gathered using RegEx as its parameters, then generates a definition of each parallel loop's execute function will be generated for each hardware platform, in the form of Kernel files. At compile time these Kernel files will be linked to the extern definition in the Modified Application File by the linker.

The requirements for the code generation script that will be created as part of this project to produce kernels containing CUDA code with JIT compilation, will discussed in the following section.

## 3.2 New System Requirements

Implementing the new system will require work in two main areas: a new Python code generation script, and some modifications to the OP2 library itself. The OP2

Library is currently implemented in both C and Fortran, but only the C library will be modified, due to developer familiarity with the C language. OP2 does also include code generation using MatLab, however the Python script is preferable for new developments, since Python is now ubiquitous, and provides very convenient string manipulation capabilities. The following are the necessary requirements to consider this project a success.

### 3.2.1 Python Script

The new code generation script will be named `op2_gen_cuda_jit.py`, and will need to perform a somewhat similar source-to-source translation process to pre-existing CUDA script for Ahead-of-Time compiled code. The extension required is the ability to also generate a second, altered code-base that will be compiled at run-time, as well as the original code that is compiled prior to running the executable.

All code generated by the new code generation script must form valid C files, and compile using a the Intel C compiler [27] without errors. Since the project will involve generation of NVidia CUDA as well, the generated CUDA code must also be valid, and compile with the NVidia C Compiler (`nvcc`) from the NVidia CUDA Toolkit [6, 5] without errors.

When the resulting executable has been compiled from the generated code is executed, it will need to invoke a re-compilation stage while it is running, and execute code that has been compiled during its runtime as part of its execution. It must produce an output within some tolerance of the expected result, obtained from executing the parallel loop iterations sequentially in an arbitrary order. The order is not significant, as OP2 enforces a restriction that the order in which elements are processed must not affect the final result, within the limits of finite-precision floating-point arithmetic [18, p3]. There cannot be any dependencies between iterations in the application, otherwise the result may vary when translated by OP2. This constraint allows the OP2 code generator the freedom to not consider the ordering of iterations, and instead select an ordering based on performance.

### 3.2.2 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", built on the assertion that values declared as OP2 constants are certainly not going to change during the course of execution. To apply this constant values provided as part of the input must be turned into `#define` directives for the C Pre-processor in the recompiled code. This will result in all references to the variable's identifier in the code being transformed seen as the literal value the variable holds by the compiler.

An example of how this is normally used can be seen in the CUDA example program from before (Appendix A), where the size of the arrays to be added together is defined as `N`, and everywhere `N` appears in the code the literal value `32` will be substituted before compilation.

As a result, the need to store constant values in memory has been removed, and when a constant value is required, retrieval time from memory has been eliminated as the literal value is immediately available. Other possible optimisations will be discussed in Section 6.1 (Future Work).

### 3.2.3 OP2 Library

Outside the code generation script, some OP2 API functions may need to be implemented differently in the OP2 library files, as the functions may require additional information to be stored and retrieved at runtime. It is a requirement that the OP2 API itself is not altered in any way by modifications to the library, so that all existing programs currently using the API will be able to seamlessly update to using the modified version.

## 3.3 Testing & Benchmarking

Once the code generation stage has been completed, and the Python Script is able to generate valid C and CUDA code that can be compiled without error for an example OP2 Application, the resulting binary needs to be tested to ensure the result is correct, and benchmarked to determine if there is performance gain. The easiest thing to compare to performance against is the same application generated for graphics cards, without JIT compilation, and see if there is any benefit. Benchmarking results will include the time taken to recompile at runtime for the JIT compiled version, and the time taken to copy constant values to device memory for the AOT compiled version.

## 3.4 New System Model

Figure 11 describes the new workflow of the OP2 library, with Just-In-Time compilation. As before, code generation takes an input of the application and optional additional files, and generates the Kernels and Modified Application Files. It also generates an additional set of Optimised Kernels, which contain code that are only be compiled inside the green box denoting 'run-time', at which point the constants from the Input Map are known to the program. These Kernel files are not seen by the ahead of time compiler.

The JIT compiler also needs to link the Optimised Kernels against the OP2 Library Files, so it is necessary that they are stored in a location that is also accessible at run-time, not just when the executable is compiled. This compilation

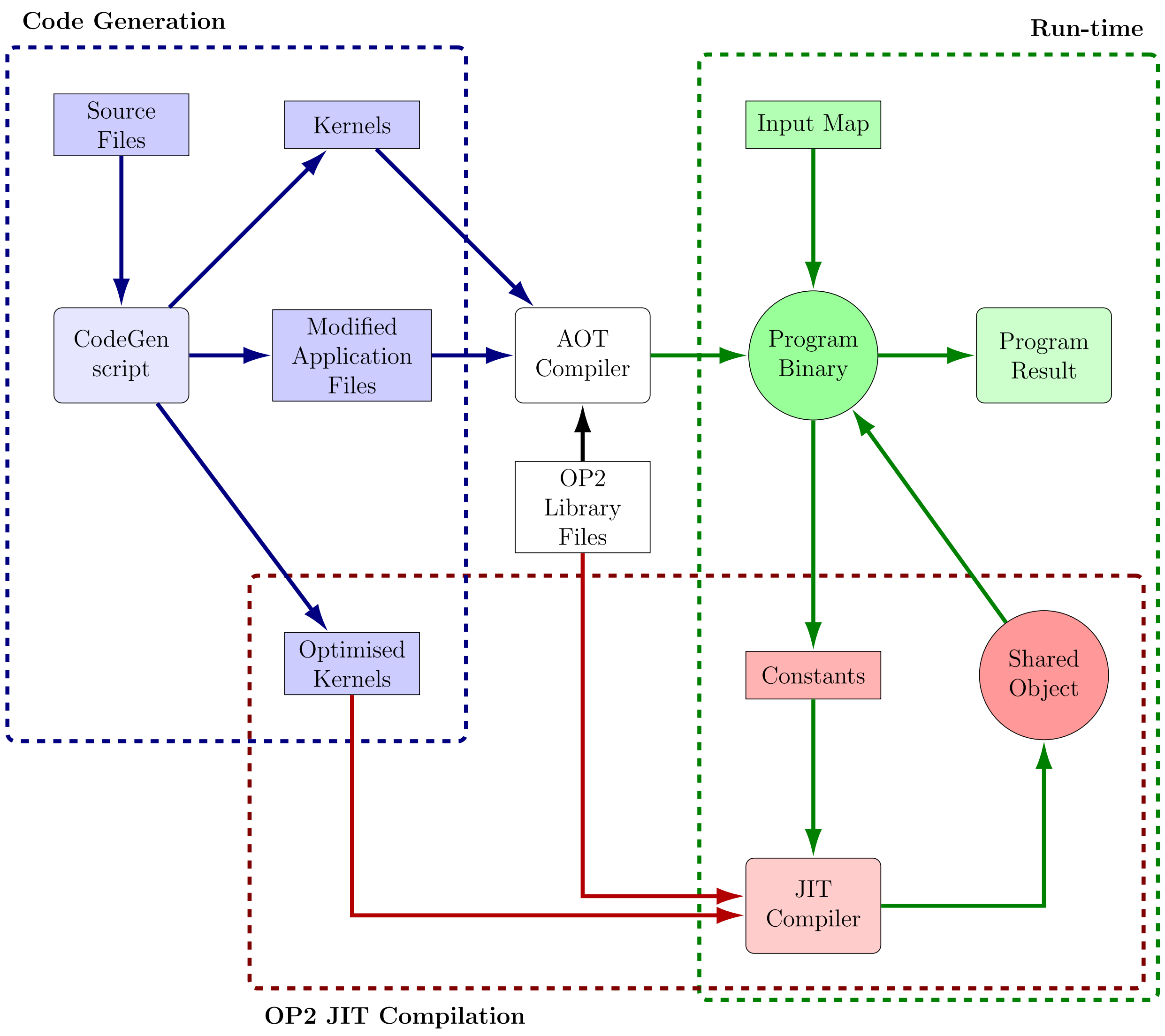


Figure 11: OP2 System Diagram with JIT Addition

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 theoretically 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 pass execution over to the JIT compiled function. It could be beneficial, however, to allow executables with the JIT compilation feature enabled or disabled to be compiled from the same source code. This requires the Ahead of Time Kernels to have retain the ability to execute the loop body without requiring JIT compilation, as well as being able to initiate the runtime compilation of the optimised kernels. Therefore the compiler invocation will be wrapped in a pre-processor conditional, so that the feature can be enabled or disabled using a compiler argument: `-DOP2_JIT`.

## 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)
```

[17, 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, it needs to maintain a de-duplicated list of identifiers, and persist their values, data types, and dimensions. These parameters will be used when the first parallel loop is invoked to generate the header 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.

In the next section, the completed Implementation is discussed. The contents of the files generated by the Python code generation script will be examined in more detail, and design decisions made will be explained. The Implementation is presented in its finished form, however the development process is covered later, in Section 6.2.

# 4 Implementation

The new optimisation of adding a compilation stage during runtime, and defining constants from the input, was implemented over the course of approximately 50 hours as part of this project. The following Section describes the completed addition to the OP2 Framework.

The source code for the existing OP2 Library is hosted open source as a GitHub[12] repository. Instructions for obtaining the version described in the following section, and for getting started with OP2, are provided in Appendix B.

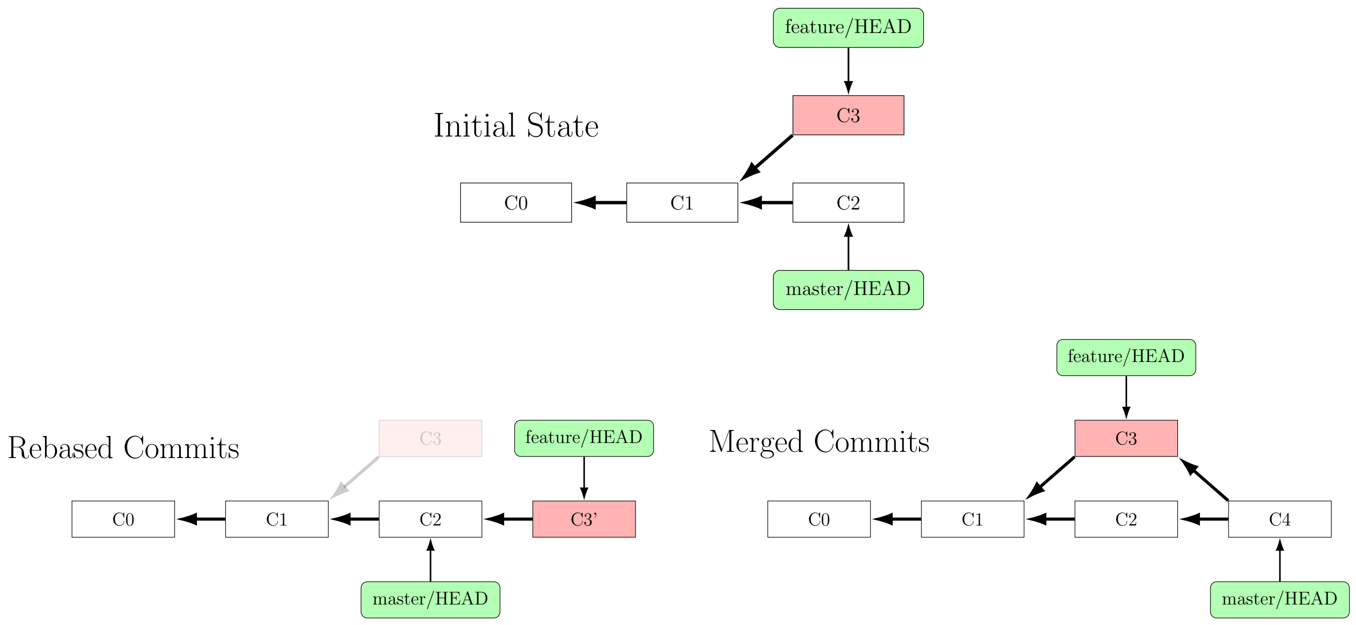
## 4.1 Git Repository

In the Git Repository, the feature branch for this project is named `feature/jit`. It was branched from `feature/lazy-execution` on 13th November 2019, which was last committed to in April 2018, and therefore it lagged behind the `master` branch somewhat. `feature/lazy-execution` needed to be synchronised with the `master` branch before any other changes could be made.

A rebase was selected as the best method of synchronisation. In git terminology, a rebase involves making copies of a branch's commits, and "re-playing" these changes on to the top of another branch [20]. 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 code-base with all the features of both branches available.

Figure 12 outlines the differences between synchronisation using `rebase`, and the more commonly used `merge`. For this project, rebasing was preferable to merging as a synchronisation action, as the result is a linear branch history, rather than

Figure 12: Rebase vs. Merge. Diagram reproduced from [20]



creating a diamond. This is a destructive action, i.e. the git history is re-written, but it allows a future viewer to easily follow the origin of each line of code. Using rebase also avoids modifying the master branch.

Furthermore, in the case of merge conflicts (where a change has been made in both branches, and one needs to be selected) a rebase will halt at the first conflicting commit and allow the conflict to be resolved [19], while synchronising using merge would result in receiving all conflicts in one go, which can make the necessary resolution of the conflicts harder, especially if not familiar with either of the branches, and which change is the newest.

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, since it was unlikely the rebase would need to be undone, and the previous state of both branches remains untouched since a new one is being created.

The `feature/lazy-execution` branch was created for developing a system to execute parallel loops when resulting 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 to save time. Lazy execution will not be the focus of this project, however this process for invoking parallel loops will continue to be utilised throughout the work done to enable Just-In-Time Compilation for CUDA, so that future efforts towards lazy execution can be continued 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 named `op2_gen_cuda_jit.py`, which is located in the folder: `translator/c/python/jit/` of the OP2 repository.

This code generator, which produces source files for CUDA with JIT compilation, is called from another Python script named `op2.py` which was explained in the Section 3.1.1. `op2.py` can be found in the parent directory: `translator/c/python/`, and its purpose is to handle the generation of the Modified Application File. Since the existing Modified Application File generation is sufficient to meet the requirements of this project, `op2.py` is only slightly changed. The modification made is adding a call to the new code generator described below.

### 4.2.1 jit/op2_gen_cuda_jit.py

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

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

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

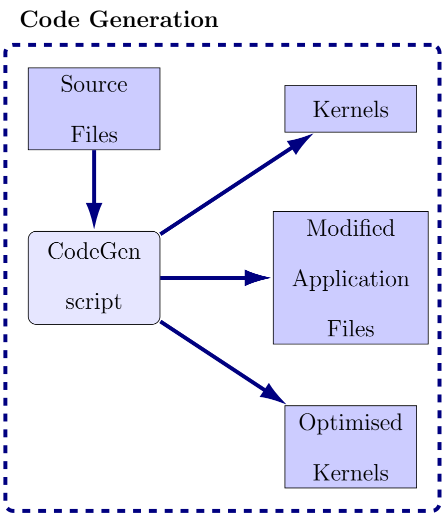
The arguments passed to it from `op2.py` are:

| | |
|---|---|
| **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. |

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`

In the JIT filename "`_rec`" is short for "recompiled". These files were referred to as "Kernels" and "Optimised Kernels" respectively in the System Model from Section 3.



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

- `cuda/[application]_kernels.cu`

It will contain function definitions required by all loops, or by the 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.

### 4.2.2 Execution Setup

The first action performed by the code generation script when it is invoked is to check across all kernels for the Struct-of-Arrays data layout, or if all are using the default Array-of-Structs. If any do use SoA, a flag named `any_soa` becomes a non-zero value, so will evaluate as `True` in a conditional.

Then, a folder `cuda/` is created if it does not 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.

### 4.2.3 Kernel Files

As mentioned above, the code generator outputs two C source code files for each parallel loop. The following section explains these kernel files, covering the purpose of each function in the order they are generated, and how they can vary based on the inputs.

To avoid ambiguity between code written by the developer, and code that has been generated as part of the output, Python code that is an extract of the implementation will be marked with just a file and line reference, and C code that has been generated as part of the output of the script will be marked *generated by ...* and then a file and line reference.

Furthermore, Python code will also always be in a frame filled grey, while generated C code will be in green, blue or red frame - depending on if the code contained in the box is unique to the JIT kernel, the AOT kernel, or is common to both.

| Python Code | JIT Kernel Code | AOT Kernel Code | Common Code |
|---|---|---|---|
| ⬜ | 🟩 | 🟦 | 🟥 |

Figures 13-18 show the progression of the two kernel files for a typical parallel loop during the execution of the code generation script (starting from empty files). They are provided only for the purpose of highlighting the relevant sections of each file. The generated code in the figures is not intended to be a legible size.

It may aid in understanding to follow this section with either the translation script, or a set of generated kernel files to hand, since full code listings are not included for every section. A summary of the generated functions can be found on page 42.

**1. JIT includes:**

The first piece of C code generated by the Python script is simply OP2 library include directives. These are only 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 AOT kernels, as they will be included by the central kernels file, which will contain these same `#include` statements:

```
#include 'op_lib_cpp.h'
#include 'op_cuda_rt_support.h'
#include 'op_cuda_reduction.h'
...
```

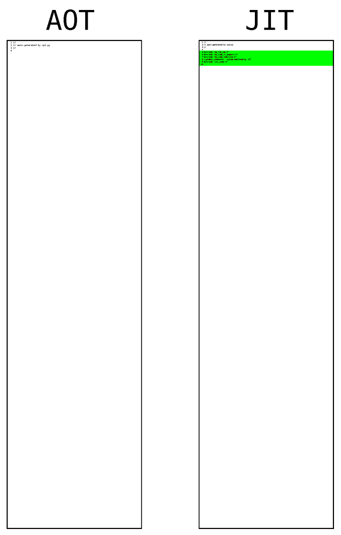
generated by TODO

The JIT kernel file also includes a file named `jit_const.h`, 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 - values #defined by JIT
#include 'jit_const.h'
```

generated by op2_gen_cuda_jit.py [170-172]

Figure 13: JIT includes



The Python code for generating these statements makes use of the `code()` and `comm()` helper functions, which automatically indent using a global variable `depth` that is updated whenever scope is changed.

```
comm('global_constants - values #defined by JIT')
code('#include "jit_const.h"')
code('')
```

op2_gen_cuda_jit.py [170-172]

**2. User Function:**

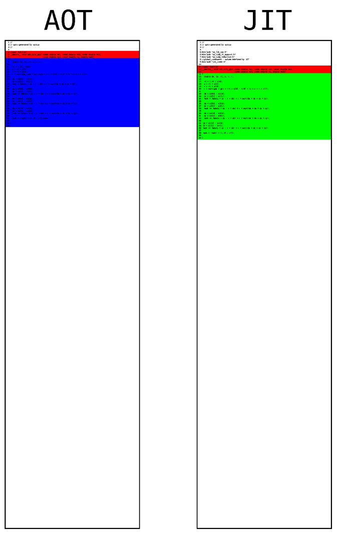
The User Function is the operation specified by the user to be carried out on each iteration of the loop. This function will run on the device (GPU) on many threads simultaneously, performing an action at least once for each set item.

The User Function is given the `__device__` function descriptor, so that it will be compiled for execution on a GPU device, and so it 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] )
{
  ...
```

generated by TODO

Figure 14: User Function



The function body is pulled from a function written by the application programmer, and found in the input files: either the Application File, or one of the optional header files. Once found, it needs to be checked to ensure it has the correct number of parameters, otherwise it is not valid to be used as the user function, and code generation will end with an error.

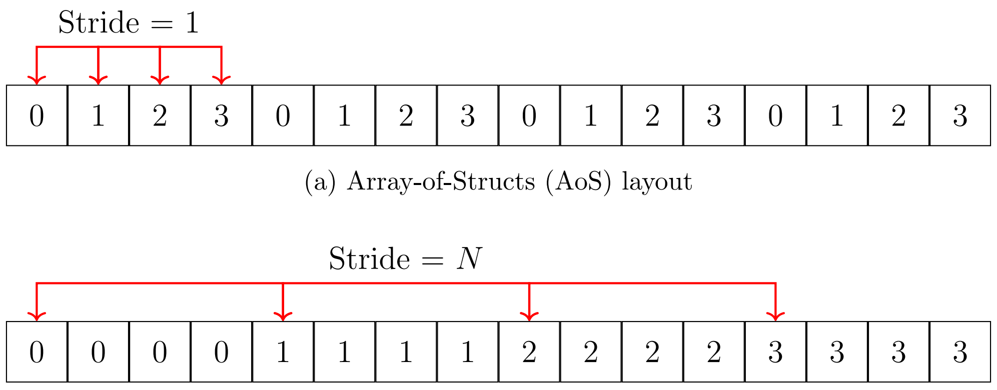
Any `#include` statements in the file containing the user function are replaced by the contents of the file, exactly as the pre-processor would do normally.

**Data Layout:** If the flag for automatic Struct-of-Arrays data layout transformation is not enabled, the function body will remain largely the same as defined by the application programmer. However, if it is enabled, there are modifications that need to be made to the function body to achieve this.

The code for making this transformation is pulled from the pre-existing AOT CUDA code generation script:

`translator/c/python/aot/op2_gen_cuda_simple.py`.

The purpose of the code segment (`op2_gen_cuda_jit.py` [242-257]) is to multiply array access indices by the stride for that data structure, which will be set as a constant later by the Host Function. If the access is indirect, it is the second index that is multiplied by the stride of the inner map.



(a) Array-of-Structs (AoS) layout

(b) Struct-of-Arrays (SoA) layout

Figure 15: Data layouts with labelled strides

**Constant Definition:** The Constant Definition optimisation needs to be applied to the user function, as it contains 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 - where the value will have been copied when it is defined as a 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, which is achieved by the following lines from the translator script:

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

op2_gen_cuda_jit.py [905-907]

**JIT:** The JIT kernel needs to be modified differently. Constants with a dimension of 1 (i.e. they contain only 1 value) can be left unchanged, as the literal value will be defined under that same identifier. There is no possibility of a name collision here since the identifier will never be allocated memory, only replaced by a literal value.

Constants with multiple values (i.e. with a dimension greater than one) cannot be defined as a macro, since macro values cannot have multiple values, and cannot be indexed. It also CUDA does not allow variables to be declared both `__constant__`, and given external linkage using `extern` [7, p126], which is how they are handled for the sequential JIT implementation.

The solution to this challenge comes 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)}
```

op2_gen_cuda_jit.py [931-934]

However, 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]    ⇒    int A = op_const_c_array_1 + 2
```

If this problem is not solved the most likely outcome is an undefined identifier error at compile time. In the above example the code will compile without error, but 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 can then 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, ..._2, }
 ...
int A = op_const_c_array[1+2]
```

generated by TODO

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 [3] to evaluate the expression at compile time where possible, but this was not done due to the unlikely possibility of such code actually being written.

The full Python listing for generating C code to handle constants in JIT compiled kernel is provided below.

```
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
```

op2_gen_cuda_jit.py [931-944 UPDATE]

**SoA optimisation:** Since the modifications to enable the Struct-of-Arrays data layout involve constant values for the stride of each data structures, an attempt to streamline this process using the Constant Definition optimisation was made during this project. It was unsuccessful, with a longer discussion later on the reasoning in a later section on the Host Function.

**3. Kernel Function:**

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

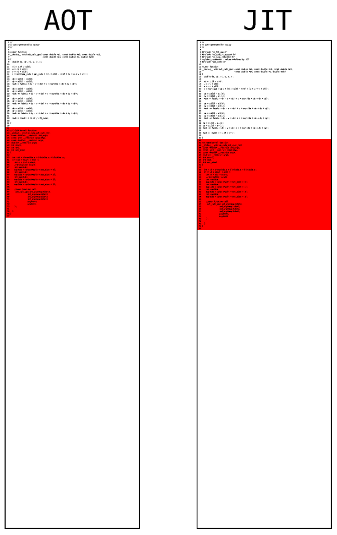
The kernel function is the same in both files, and is also executed on the GPU across all the parallel threads. It is declared `__global__` so that is executed on the device, but can be called from host (CPU) code:

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

generated by op2_gen_cuda_jit.py [TODO]

The purpose of this function is to use the CUDA built in variables `threadIdx.x`, `blockIdx.x`, and `blockDim.x` to map a unique portion of the workload onto each executing thread.

Figure 16: Kernel Function



**Indirection:** If the loop is indirect, and uses values from another map as indices, 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 already formatted in the manner it expects to receive it. It is possible that the indirect map is optional, in which case the `optflags` argument needs to be 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 requires, followed by performing any data reductions necessary. The supported reductions are: sum, maximum, and minimum [17, 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 in parallel on the GPU. The function body is the same for both AOT and JIT: setting up function arguments, block and thread sizes for the CUDA call, and timers to record how long is spent in each parallel loop. The head of the function does differ however, as highlighted in Figure 17.

Figure 17: 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 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, as it would be part of the lazy execution feature, but currently the loop will simply be called immediately after it is queued.

At the top of the function a decision is made as to whether JIT compilation should be used, based on whether the pre-processor flag: `OP2_JIT` has been defined. This allows JIT compilation to be enabled by passing the compiler argument `-DOP2_JIT`, and otherwise by default it will be disabled. If JIT compilation is enabled, then the compiler is invoked if this execution is the first of the application, then the pointer to the newly compiler version of the function is executed instead.

The actual invocation of the compiler process is handled by the `jit_compile()` function, which has not yet been generated, as it will reside in the central kernels file. It will be discussed in detail, along with the other functions in that file, in Section 4.2.5.

If JIT is not enabled, the lines of code between `#ifdef` and `#endif` will be ignored by the compiler, so the process will continue into the AOT host function, which causes it to stay within the AOT kernel file and never execute any code from the JIT file.

The pre-processor condition section is generated using the following Python code:

```
    code('#ifdef OP2_JIT')
    depth += 2
    IF("!jit_compiled")
    code('jit_compile();')
    ENDIF()
    code('(*'+name+'_function)(desc);')
    code('return;')
    depth -= 2
    code('#endif')
    code('')
```

op2_gen_cuda_jit.py [546-555]

**JIT:** The code generated for the top of the Host Function in a JIT kernel file is shown below. Contrasting with the code generated for the AOT kernel file there are a few key differences. Firstly, since this function needs to be linked to the existing 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 [11].

```
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 op2_gen_cuda_jit.py [522-531]

As can be seen above, the function also has a different signature:

```
op_par_loop_[name]_rec_execute(op_kernel_descriptor* desc)
```

Instead of:

```
op_par_loop_[name]_execute(op_kernel_descriptor* desc)
```

As before, “rec” is short for **recompiled**. This version of the function 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, by the following line from the code listing on the previous page:

```
      (*[name]_function)(desc);
```

Since it resides in the JIT kernel file, it makes calls to the Kernel Function and User Function in the same file as itself, rather than those in the AOT file, and as such the optimisations made to the User Function in the JIT kernel file are able to be used.

**SoA:** If the Struct-of-Arrays Data layout is enabled, the body of this function in both AOT compiled and JIT compiled kernels will need to set the stride length for each data structure and copy it to a CUDA device symbol. This is only done on the first iteration of each loop.

```
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 op2_gen_cuda_jit.py [640-654]

These sizes only become available when the function has been called and it's arguments are known. As previously mentioned, an attempt was made to replace these constants with defined literal values in the JIT kernel, however this proved difficult as each loop would need to have been called at least once so that all the strides are known before the compilation could be done.

For this to be possible, each loop would need to execute correctly before JIT compilation as well as after in the binary with JIT compilation enabled, and therefore all the input constants would need to be copied to device memory as usual, otherwise they would not be available before the re-compilation is initiated. This would add extra duration to the upfront cost of the optimisation.

Furthermore, parallel loops may not all be created equally. It is a possibility that a certain loop never gets called, or is only called for the first time half way through an application. In a situation such as that, any benefit that could be gained from JIT compiling would be wasted while waiting for every loop to have been called at least once.

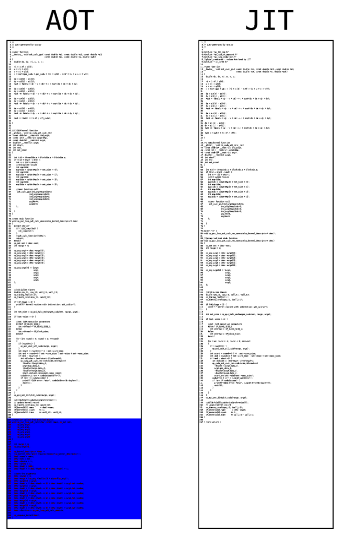
For this reason, the data structure strides remain as a device constant which is copied to device memory on the first iteration of a loop in both AOT compiled and JIT compiled kernels

**5. Loop Function:**

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

The Application File will be modified by `op2.py` to contain an declaration for this function marked `extern`, to be linked against this definition in the CUDA version of the executable. Only the AOT kernel requires this function, as the Host Function acts as the entry point for the JIT compiled kernel The function signature is:

Figure 18: Loop Function



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

generated by op2_gen_cuda_jit.py [878]

The purpose of the Loop 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, and is passed as an argument to the enqueue function so it can be executed at a time decided by the lazt execution subsystem:

```
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 summarise, for every parallel loop two separate kernel files containing C and CUDA code are generated, one for Ahead of Time compilation, and one for Just in Time compilation. The generated code in a particular pair of kernel files will to be executed when the corresponding loop is invoked in the Application File, which has been modified so that the compiler will link its function calls to the function definitions generated above.

Figure 19 has been included to clarify the data flow through the two files, where an arrow from Function A to Function B indicates that B is called from the body of A. The diagram starts with the Loop Function at the bottom which is called from the Application File, and executed by a single thread on the host. The AOT Host Function is eventually called, where either the re-compiled JIT version is invoked, or the original version is used if JIT compilation is not enabled when the application is first compiled. In the host function the GPU device is configured, and Kernel Function is executed simultaneously on many parallel threads. Each thread in turn executes the User function to perform an operation on elements of a set, as per the design of the application programmer.

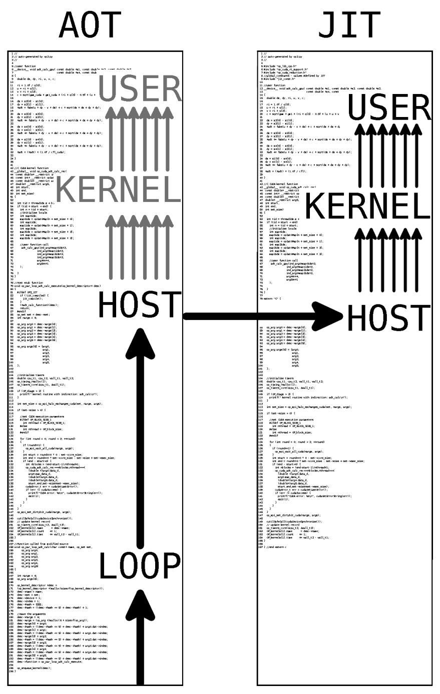


Figure 19: Kernel Flow

The `jit_compile()` function, which will handle actually performing a compilation stage at runtime, 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 resides in the `cuda/` directory alongside the Kernel files. It is named: `cuda/[application]_kernels.cu` and is the final source file to be generated, once the Kernel files for each parallel loop are complete. It will tie up the remaining loose ends, as it contains the `jit_compile()` function for invoking the run-time compiler, and the definition for the new OP2 API function for declaring constants.

It also contains `#include` statements for each of the AOT kernel files, so that their contents is imported to make a single file, and can be compiled in a single parse. 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 required OP2 library files, as seen at the very start of this Section for the JIT compiled kernel:

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

generated by TODO

As mentioned in that section, these serve as a reference to the OP2 library files for all of the AOT kernels, and their inclusion here is the reason the AOT kernels do not each individually require these statements.

A declaration of a CUDA constant for each input constant in the user's dependingn is generated next. An example of what this might look like is shown below.

```
__constant__ double single_cuda;
__constant__ double value_cuda;
__constant__ double constant_cuda;
__constant__ double four_vals_cuda[4];
```

generated by op2_gen_cuda_jit.py [1026-1037]

These declarations are 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+'];')
```

op2_gen_cuda_jit.py [1026-1037]

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 when the programmer wishes to declare a constant identifier and value in the input; 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 retrieved and 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 [17, p9]:

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

This signature will not be altered to ensure backwards compatibility. Fortunately, it does not need to be modified for the requirements of this project, so this is not an issue.

Two versions of the function are generated, but only one will be needed depending on whether JIT compilation is enabled or disabled. To achieve this, the two function 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.

```
#ifndef OP2_JIT

void op_decl_const_char(int dim, char const *type,
                        int size, char *dat,
                        char const *name)
{
  ... //JIT disabled function definition
}

#else

void op_decl_const_char(int dim, char const *type,
                        int size, char *dat,
                        char const *name)
{
  ... //JIT enabled function definition
}
...
void jit_compile() {
  ...
}

#endif
```

The `jit_compile()` function is also wrapped by the pre-processor conditional, as it is only going to be required if JIT compilation is enabled. Although it would not detriment the program to include it in both versions, it is not necessary.

**AOT:** The top version of the function, for when JIT compilation is disabled, is based on the existing code generation, and therefore copies the constant value passed to it to the device constant using a CUDA function for moving a value between host memory and device memory:

```
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.

**JIT:** The JIT version instead invokes the OP2 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]

This function was added with lazy execution, and maintains a de-duplicated list of constants, so that once they all have been declared the header file defining each value can be generated. As can be seen in the generated C code below, constants containing more than one value are iterated over, and each element is declared as asingle value due to the issues with `extern __constant__` values described in Section 4.2.3 (2. User Function).

```
#else

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 op2_gen_cuda_jit.py [1092-1114]

**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 OP2 library functions for timing which kernel files used previously to gather data on the time spent in each parallel loop, as the time spent re-compiling the binary is important for performance comparisons later.

Above the `jit_compile()` function, in global scope, a function pointer is declared for each parallel loop, and defined `NULL` so that after re-compiling the new version of the function can be referenced using the function pointer.

```
code('')
comm(' pointers to recompiled functions')
for nk in range (0,len(kernels)):
  name = kernels[nk]['name']
  code('void (*' + name +\
       '_function)(struct op_kernel_descriptor *desc) = NULL;')
```

op2_gen_cuda_jit.py [1016-1120]

The output of this Python code is a number of lines of C with the following form:

```
// pointers to recompiled functions
void (*[name]_function)(struct op_kernel_descriptor *desc) = NULL;
void (*[name]_function)(struct op_kernel_descriptor *desc) = NULL;
void (*[name]_function)(struct op_kernel_descriptor *desc) = NULL;
...

void jit_compile() {
```

generated by op2_gen_cuda_jit.py [1016-1120]

**Invoking the Compiler:**

As can be seen below, the compiler is invoked by the executable through a system call to initiate a GNU Make [26] command. It is expected that the Makefile is accessible at runtime as well as at compile time, and that it contains a target named `[application]_cuda_rec` which will perform the necessary compilation. Furthermore, the compiler arguments, library install paths, and other parameters of the compiler are handled by the Makefile. The contents of the Makefile for this implementation will be covered in Section 4.3.

Once the compilation has completed, the terminal output of the command is stored in a log file in case of an error, which will also cause an error message is printed, and the program to exit early.

This is the C code generated to perform the compilation, with error checking.

```
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 op2_gen_cuda_jit.py [1139-1146]

It is expected that the result of the compilation will be a shared object file named `cuda/airfoil_kernel_rec.so`, which exports functions for each parallel loop. If this file does not exist the application binary will exit with an error, otherwise the recompiled function for each parallel loop is dynamically loaded into the application, using: `void *dlsym(void *restrict handle, const char *restrict name)` from *dlfcn.h*.

For every parallel loop, the function `op_par_loop_[name]_rec_execute` is imported, with its address stored in the void pointer declared in global scope: `[name]_function`. We have seen this pointer before, in Section 4.2.3, where is was used to call the JIT kernel Host Function from the AOT kernel Host Function.

The value return from `dlsym()` needs to be cast to the type of the function signature:

`(void (*)(op_kernel_descriptor *))`

```
//dynamically load functions from the .so
[name]_function = (void (*)(op_kernel_descriptor *))
                  dlsym( handle, "op_par_loop_[name]_rec_execute" );
if ((error = dlerror()) != NULL) {
  fputs(error, stderr);
  exit(1);
}
...
```

generated by op2_gen_cuda_jit.py [1160-1169]

Once all the exported functions from the shared object file have been imported, the wall clock time since the start of the `jit_compile` function is printed to the terminal. The time will be used later when analysing the optimised application runtime.

The final part of this Section on the project Implementation is on the Makefile. While this is not generated by the Python script, and would need to be recreated by an application programmer, it was part of the process to create and therefore needs to be covered.

## 4.3 Makefile

This implementation relies on GNU Make to control Ahead of Time compilation, as well as Just in Time compilation during runtime. This includes setting parameters that will be passed to the compiler process, and other options. A number of other libraries were required to build an OP2 binary even before JIT compilation was added, as described in Appendix B - *Getting Started with OP2*, so here only the recompilation target will be discussed, as it was the target developed for this project.

The binary expects there to be a Makefile in the directory it is executing in, with a target: `[application]_cuda_rec` that controls re-compilation, 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 exports the recompiled loop functions.

The library object is produced by compiling each of the kernels individually, using the NVidia compiler `nvcc` as the code contains CUDA code, then linking them into a single object. It is necessary that the compiler flags include `--compiler-options -fPIC`.

The flag --compiler options passes a list of arguments to the underlying C compiler which handles all non-CUDA sections of the code, in this case passing the argument -fPIC, which is for forcing generation of Position Independent Code. The result is that the library function will execute correctly regardless of the address at which it is loaded in memory, which is important for dynamically loaded functions.

The target cuda/airfoil_kernels_cu.o, which [application]_cuda_rec has a dependency on, is also declared PHONY, so that it is always recompiled even if the file is already considered up to date. This is so that the code is forcibly re-compiled with the pre-processor flag in the new state, even if the code has not been modified, otherwise the make flag would not function correctly. A JIT enabled version of this file may be compiled even if the variable is set to FALSE, if the kernel files are skipped on the assumption that they are up to date.

### 4.3.1 Optional Functionality

By default, the JIT compilation functionality is enabled in the Makefile provided in apps/c/airfoil/airfoil_JIT/dp/, since the default value of the macro variable $JIT is 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
```

If the JIT variable does match the string: TRUE, a compiler argument is added for the C and CUDA compilers to define OP2_JIT for the pre-processor. Additionally, the string "_jit" will be appended to the name of the executable generated, so it will not overwrite a binary with JIT compilation 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. The application 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 [16].

## 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 21a (p57). 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 [21]) 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 21b 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 21c 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 21c . 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 21d, 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 21d . 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 20: Console Output from both binaries

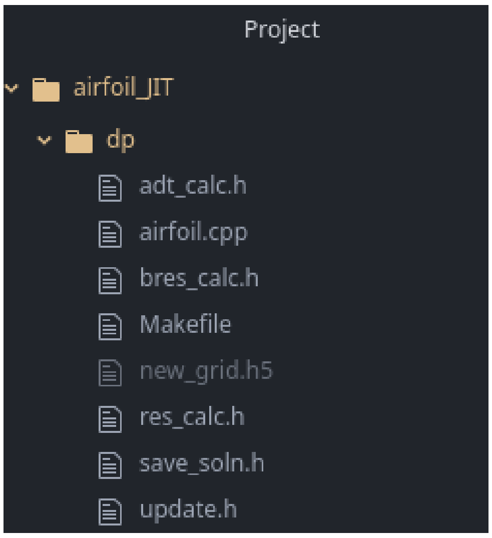
| JIT | | Enabled | Disabled |
|---|---|---|---|
| Iterations | 100 | $5.02186 \times 10^{-4}$ | $5.02186 \times 10^{-4}$ |
| | 200 | $3.41746 \times 10^{-4}$ | $3.41746 \times 10^{-4}$ |
| | 300 | $2.63430 \times 10^{-4}$ | $2.63430 \times 10^{-4}$ |
| | 400 | $2.16288 \times 10^{-4}$ | $2.16288 \times 10^{-4}$ |
| | 500 | $1.84659 \times 10^{-4}$ | $1.84659 \times 10^{-4}$ |
| | 600 | $1.60866 \times 10^{-4}$ | $1.60866 \times 10^{-4}$ |
| | 700 | $1.42253 \times 10^{-4}$ | $1.42253 \times 10^{-4}$ |
| | 800 | $1.27627 \times 10^{-4}$ | $1.27627 \times 10^{-4}$ |
| | 900 | $1.15810 \times 10^{-4}$ | $1.15810 \times 10^{-4}$ |
| | 1000 | $1.06011 \times 10^{-4}$ | $1.06011 \times 10^{-4}$ |
| 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**.

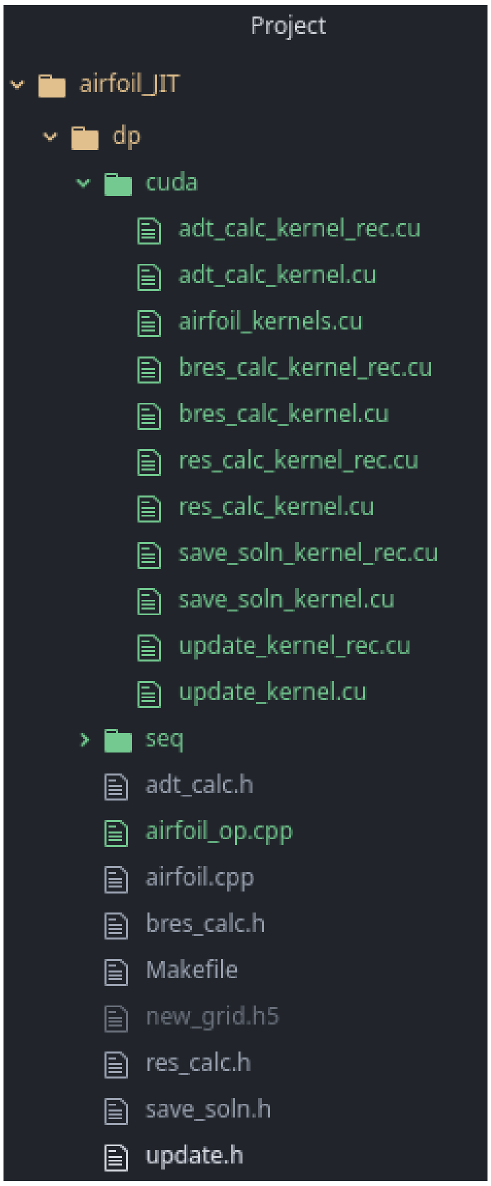
Figure 21: Files in Application Folder

(a) Input Files



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

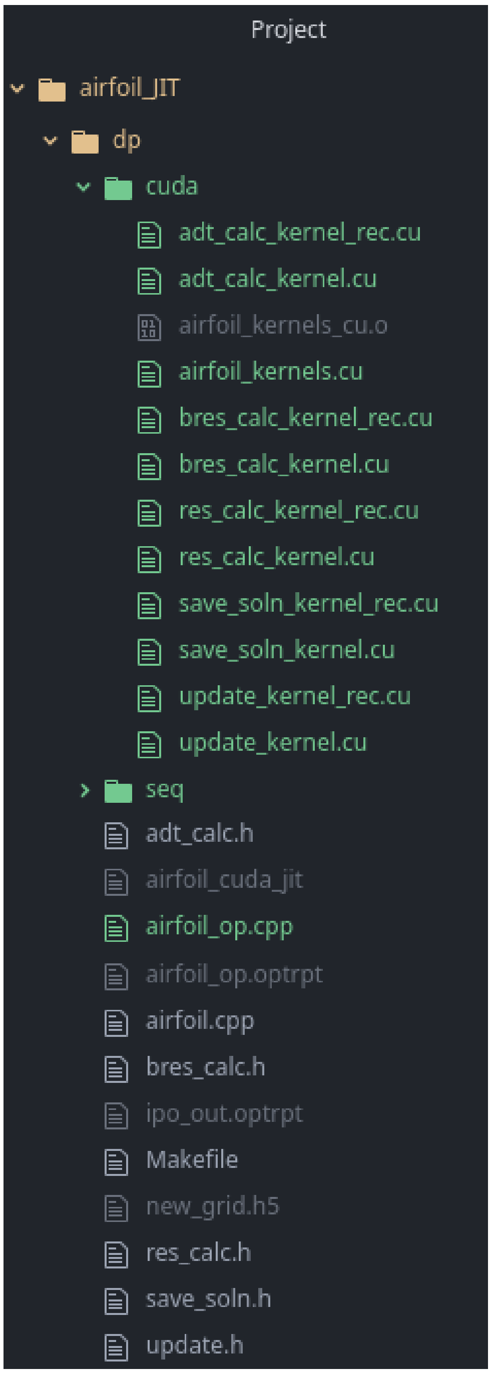
(b) After Code Generation



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

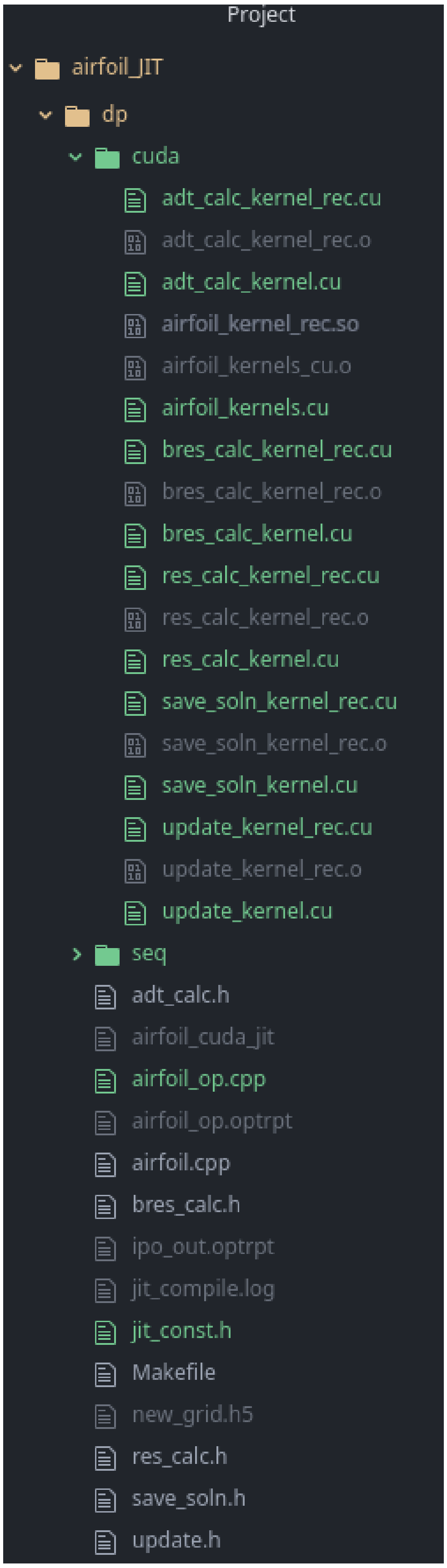
Figure 21: Files in Application Folder

(c) After AOT Compile



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

(d) After JIT Compile



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

## 5.3 Benchmarking

Once the functionality has been confirmed to work as intended, the technique can be benchmarked to determine if there is benefit in using it for run-time efficiency. Testing was done on a personal computer with an NVIDIA GeForce MX250 Graphics Card [8] - 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 [9]. 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 [37].

### 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.

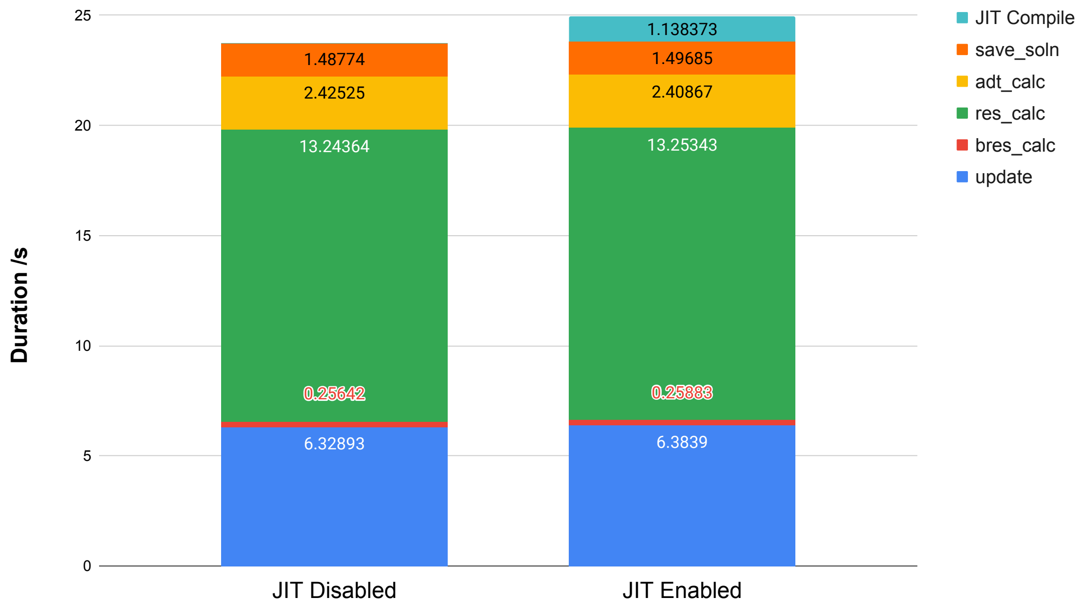
As seen previously in Section 5.2.3 (Just-in-Time Compilation), the time taken for the invocation of the compiler at run-time to complete is also recorded. It is a one-time cost at the start of execution, but still needs to be considered.

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 22 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 22: 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 22 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 [7, 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.

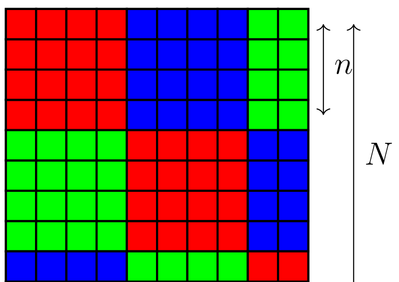
Furthermore, it is useful to discover that the technique of defining constants for the preprocessor does not sufficiently reduce the run-time duration to justify the run-time re-compilation. This will inform future investiagations into what techniques should implemented.

## 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 separate 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 23: 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 23). Doing so prevents values from being evicted from L1 cache prior to being needed again.

Currently this has only been applied to OPS [25], the precursor to OP2 [23], 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 [4], 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 69.

The Gantt chart in figure 24 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.

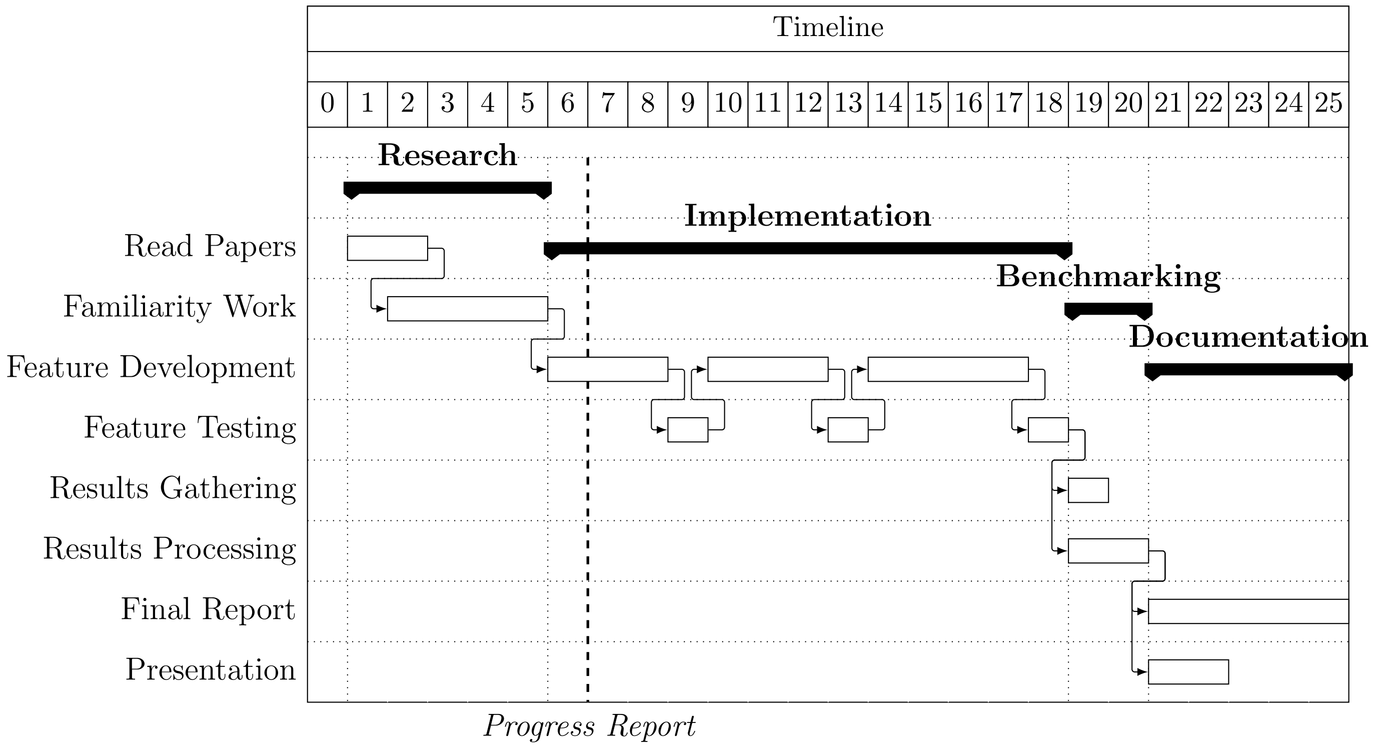


Figure 24: Gantt Diagram produced with Progress Report

### 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 found 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 24).

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] *Compiler Optimisations: Constant Folding.*
URL: http://compileroptimizations.com/category/constant_folding.htm.

[4] coneoproject. *SLOPE.*
URL: https://github.com/coneoproject/SLOPE (visited on 04/16/2020).

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

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

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

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

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

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

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

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

[13] Denis Foley. *NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data*. 2014. (Visited on 04/26/2020).

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. English. Version 3.1. 836 pp.

[15] 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).

[16] 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).

[17] 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).

[18] 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.

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

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

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

[22] D. Giles I. Reguly et al. *The Volna-OP2 tsunami code.* 2018.
URL: https://www.geosci-model-dev.net/11/4621/2018/gmd-11-4621-2018.pdf (visited on 11/05/2019).

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

[24] 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).

[25] 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).

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

[27] *Intel C Compilers.*
URL: https://software.intel.com/en-us/c-compilers (visited on 04/28/2020).

[28] *javac - Java programming language compiler.*
URL: `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html` (visited on 04/27/2020).

[29] 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.

[30] 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).

[31] 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).

[32] *OP-DSL Website.*
URL: `https://op-dsl.github.io/` (visited on 11/05/2019).

[33] *re — Regular expression operations.*
URL: `https://docs.python.org/2/library/re.html` (visited on 04/28/2020).

[34] I.Z. Reguly et al. "Acceleration of a Full-scale Industrial CFD Application with OP2". In: (March 2014).

[35] I.Z. Reguly et al. "The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations". In: November 2014. DOI: `10.1109/WOLFHPC.2014.7`.

[36] Andreas Schäfer and Dietmar Fey. “High Performance Stencil Code Algorithms for GPGPUs”. In: *Procedia CS* 4 (December 2011), pp. 2027–2036. DOI: `10.1016/j.procs.2011.04.221`.

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

[38] *The OpenMP API specification for parallel programming.* URL: `https://www.openmp.org/` (visited on 04/27/2020).

# 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;
}
```

**Compilation:**

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

**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

  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 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
```

## B Getting Started with OP2

## C Time Investment