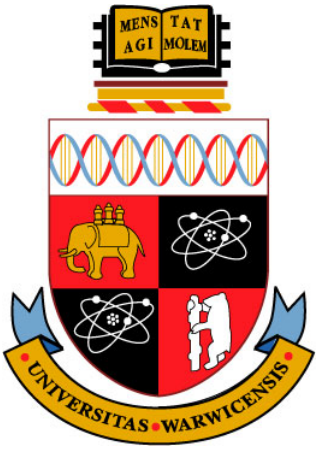


Just-In-Time Compilation for a High-Level DSL

Nathan Dunne

1604486

3rd Year Dissertation Project

Supervised by Dr. Gihan Mudalige

Department of Computer Science

University of Warwick

2019–20

Abstract

The OP2 Domain Specific Language was originally developed 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 allows HPC applications to re-compile at run-time when the inputs to the program are known. The inputs being fixed allows for more aggressive optimisations to be applied to the program, which would not be possible at compile-time.

It also covers benchmarking data gathered using the new optimisation on a representative example application. The results show that if the problem size is sufficiently large, there can be benefit, however the speed-up is not significant. Further run-time optimisations are discussed that could provide further speed-up. The finished JIT compilation platform will aid in adding additional optimisations in the future.

Key Words

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

Acknowledgements

I am grateful for the assistance given by my project supervisor, Dr. Gihan Mudalige, in helping me develop the idea for this project, and guiding me throughout with his expertise in the field. This gratitude should extend to all teaching staff at the University of Warwick Computer Science Department, for inspiring me throughout my degree.

Special thanks must also go to the Cambridge Service for Data-Driven Discovery , for allowing me to use their machines to gather benchmarking data for my implementation.

Finally, I wish to acknowledge the help provided by Yvette Dunne, Rachel Dunne, and Kaviyana Sitartha in reading over many pages of drafts, and providing much needed feedback.

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 systems almost always utilize parallelism, which involves multiple processing units working simultaneously on a single problem, to find solutions within a tractable 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 compute-heavy workloads to a GPU device can lead to significant speed-ups over sequential execution, and remove the CPU as a bottleneck.

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 implementing and benchmarking a new optimisation to the GPU code generation in OP2. The optimisation is named "Just-In-Time (JIT) Compilation" for its similarities to a comparable process often performed by compilers when run-time efficiency is desired. All compilation that takes place prior to run-time is referred to as "Ahead-Of-Time (AOT) Compilation".

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 will 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 such as in Figure 1.

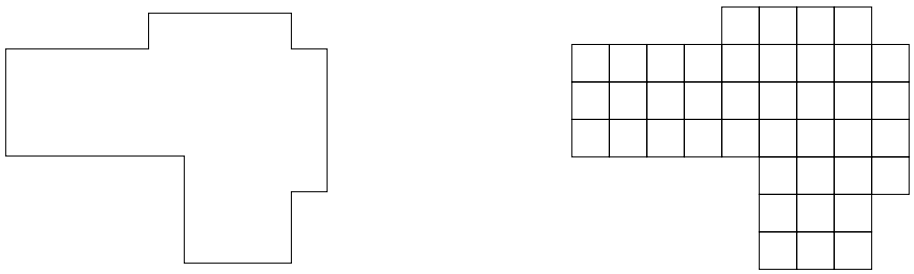


Figure 1: Example 2D decomposition

Depending on its structure, a mesh can be described as either structured (regular) or

unstructured. Structured meshes such as in Figure 2 are made up of cells following a regular pattern, while unstructured meshes use connectivity information to specify the mesh topology, as in Figure 3.

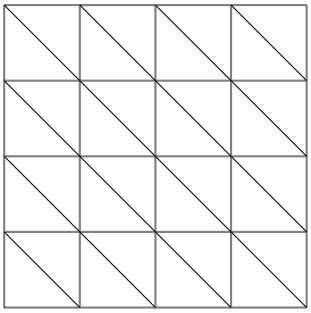


Figure 2: Tri-Structured Mesh

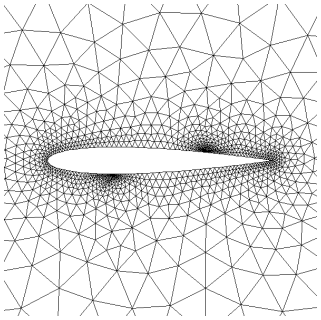


Figure 3: Airfoil Tri-Unstructured Mesh

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. A program utilising a structured mesh will make use of a repeating pattern to calculate values, which is commonly referred to as a 'stencil code' [44]. An example of a 2D stencil is shown in Figure 4.

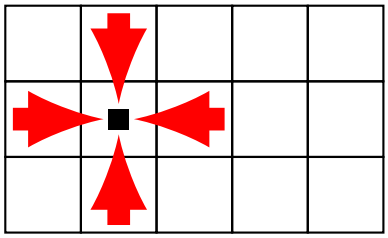


Figure 4: Example 2D stencil code

Unstructured meshes cannot use a stencil as the underlying mesh is irregular, so instead they must use indirect array accesses where a value retrieved from one array is used to index another array. When a value is updated, multiple levels of memory reference indirection will be involved, as a list of neighbouring points must be obtained, and their values loaded [2, p10]. This is much more computationally intensive that accessing data using a stencil.

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 (p17), 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 (p26) details the Implementation itself, and the expected results of code generation with JIT compilation enabled, and Section 5 (p52) explains the plan for, and outcome of, Testing and Benchmarking using an example application on a HPC cluster.

Section 6 (p72) contains an Evaluation of the work completed, including a discussion of Future Work (p72) which could build on top of what was done for this report; and a reflection on Project Management (p75). Lastly, Section 7 (p80) provides an overall Conclusion.

2 Research

2.1 OP2

This project is focussed on a contribution made to the OP2 Framework, an "active library" [24] for developing unstructured mesh solver applications. There is a large quantity of literature available on the OP2 website [38], but this section aims to provide enough understanding for someone unfamiliar with OP2 to follow the rest of this report.

2.1.1 Existing Work

The existing workflow of OP2 is as follows: it takes as application code, written in either C or Fortran with the embedded OP2 Application Programming Interface (API), and uses source-to-source translation to produce multiple different source codes, each targeting a different optimised parallel implementation. The generated source codes can then be compiled, and linked against the corresponding 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 of OP2, the journal article *Designing OP2 for GPU architectures* [36] 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. The code generation process has been modified

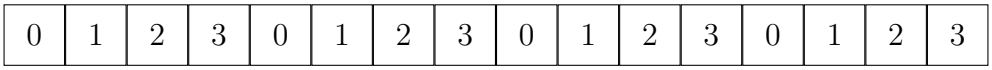
since publication, and new NVidia architectures have been released, however the article still provides useful information.

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, or the result may be incorrect. Solutions to this problem 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 [12, 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 operations are performed as a single atomic action, and therefore they cannot overlap.

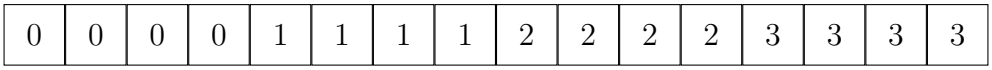
Since OP2 enforces that the order in which the function is applied to the members of the set must not affect the final result [23, p4], some data dependencies between iterations do not need to be considered by the generated code, and therefore set elements can be processed in any order, based on best performance.

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 is 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%.

The SoA layout is enabled by setting the value of an environment variable: `OP_AUTO_SOA=1`. It must be set prior to code generation [23, p13], as it modifies the output of the code generation stage, and has no effect at compile-time or run-time.



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



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

Figure 5: Data layouts. Diagram reproduced from [36]

In the Implementation section (Section 4) the difference 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 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 application input. This project aims to bridge this gap, and determine if there is benefit to be gained from such optimisations.

2.1.2 OP2 Applications

There are a number of industrial applications that have already been implemented using the OP2 active library framework, which would immediately benefit from further optimisation of the generated code, including: *airfoil* [22] - a non-linear 2D inviscid airfoil code; *Hydra* [29] - Rolls Royce's turbo-machinery simulator; and *Volna* [27] - a finite-volume nonlinear shallow-water equation solver.

They all 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.

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 updated to provide support for 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 for other OP2 applications.

2.1.3 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* [42]. This paper compares the newer OP2 framework with its predecessor OPS [43], as well as benchmarking the application against OP2 code generated utilising OpenMP [46] and MPI [20] 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 this speed-up from using a GPU is a very significant result. Hopefully this project can improve it even more, with the addition of Just-In-Time compilation.

2.2 NVidia CUDA Programming Model

Since the automatic generation of GPU code source files is a core part of this project, it is important to first 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* [12] has been summarised below, and it will continue to be used for reference throughout this report.

2.2.1 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 6, 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.

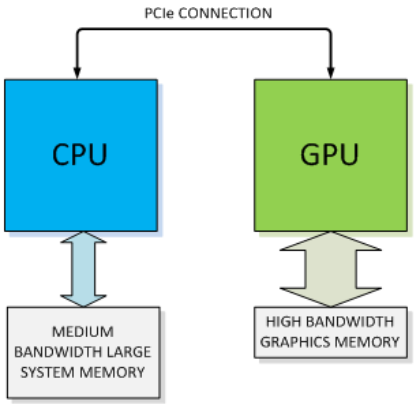


Figure 6: CPU - GPU communication. Diagram adapted from [19]

Figure 7 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 is clear why the GPU is ideal for compute-heavy workloads, where the ratio of arithmetic operations to memory operations is high.

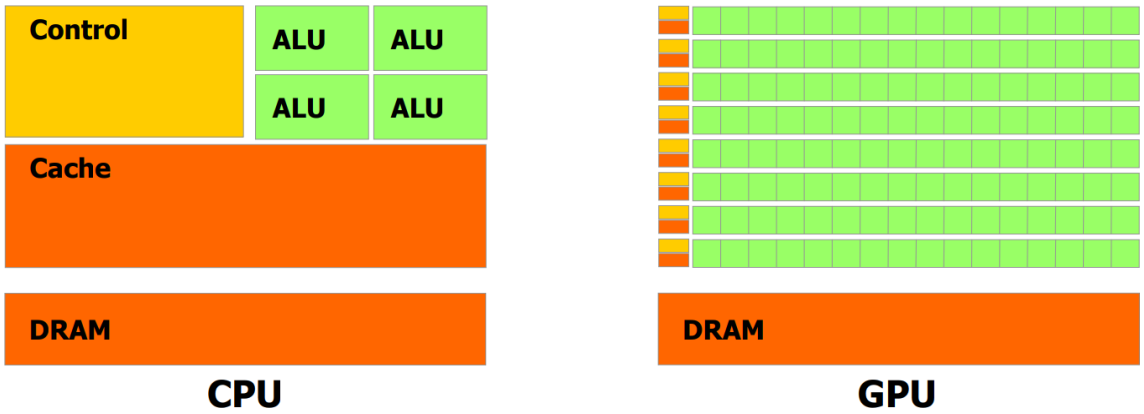


Figure 7: Architecture Comparison. Diagram from [12, p3]

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

2.2.2 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 [12, p9]. Figure 8 shows an example where 2D identifiers are used.

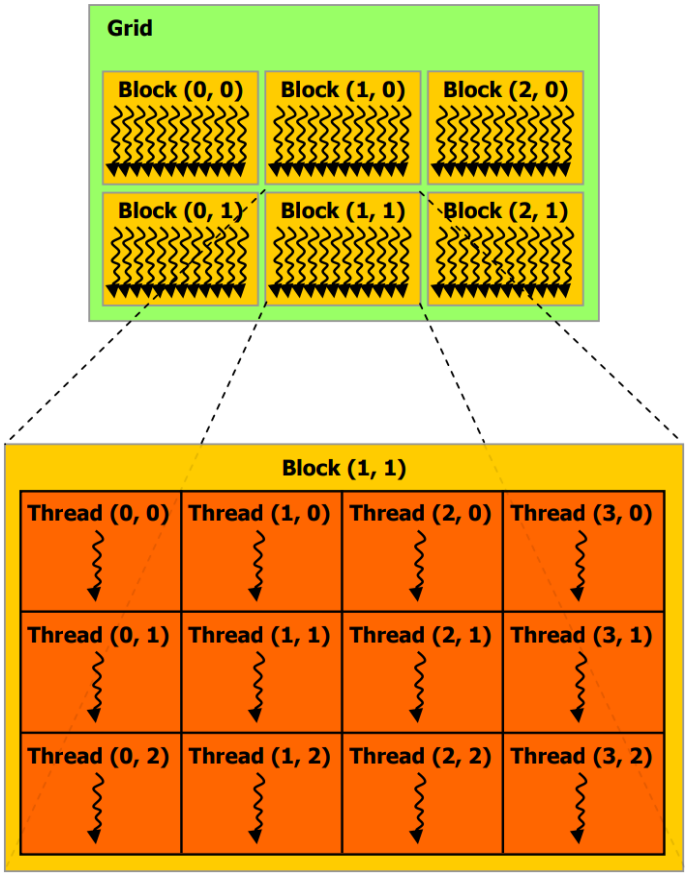


Figure 8: 2D grid of Blocks and Threads. Diagram from [12, p9]

2.2.3 Programming Interface

When using the CUDA C API, functions that will initiate the GPU are called using additional "execution configuration" syntax [12, p7], 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...] )
```

The data type of `num_blocks` and `threads_per_block` can be a normal `int` type (1D), or use a CUDA type `dim3` [12, p9] to specify up to 3 dimensions. Any value left unspecified is initialised as 1 [12, p87]. The function body will then be executed `num_blocks` × `threads_per_block` times, with all threads beginning at the same time.

There is an upper limit on the number of threads depending on the hardware, for

example the Kepler Architecture can support 2048 total threads per multiprocessor [15], 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.

Function Quantifiers

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 [12, p15], and executed on an NVidia GPU device. The difference, however, is that a function declared `__global__` can be invoked from host (CPU) code using the syntax above; whereas a `__device__` function can only be called from code that is already executing on the device [12, p81].

In the next section the new optimisation that will be applied to the existing OP2 code generation for GPUs will be discussed, along with similar implementations from other areas of the computer science field.

2.3 Just-In-Time Compilation

"Just-In-Time Compilation" refers to the process of performing some compilation at run-time when the inputs to the program are known, and can be used to optimise the application. It is called "Just-In-Time" as it is the last opportunity to optimise the application before it is actually executed.

The optimisation itself is not specific to the GPU architecture, and could be applied to any of the platforms supported by OP2, however, since GPUs are leading on performance at the moment (see Section 2.1.3), they are the most commonly used and therefore any benefit to that back-end could impact more actual users. It would also a be good indicator that JIT compilation does have potential across all platforms if it is able to provide speed-up to the GPU back-end.

The term "Just-In-Time Compilation" is most commonly associated with the Java programming language [39], 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).

2.4 Java

The Java compiler (`javac`) compiles code into platform independent "bytecode" [33], 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 [33], 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 [37] contains further detail on the JIT compiler and its impact on the performance of Java.

This core idea of recompiling "hot" 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 approach taken by Java 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.

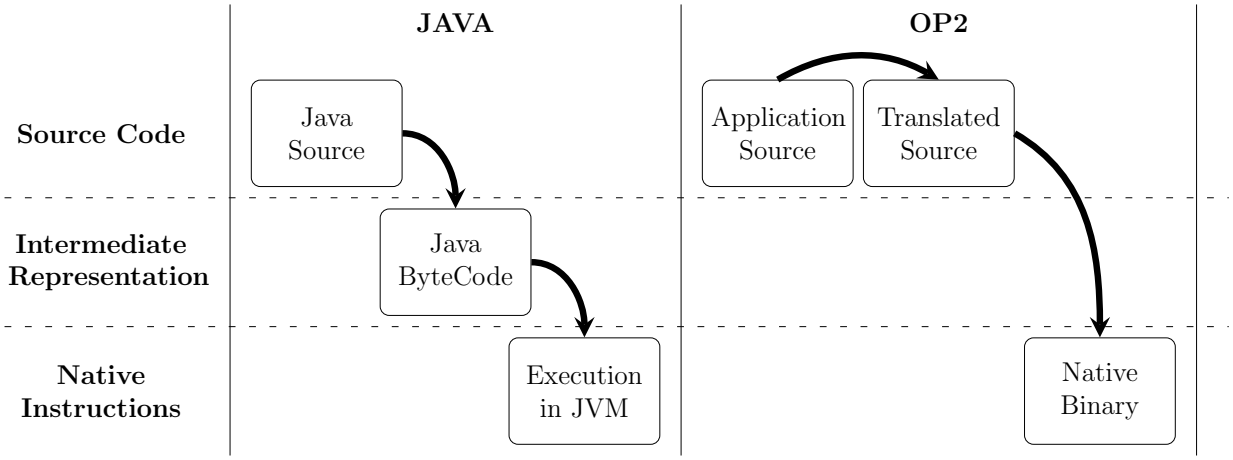


Figure 9: Comparison of Java and OP2 JIT

2.5 Related Work

While Java's popularity and success from using JIT compilation gives a good indication that there is real performance benefit to be gained from using the technique, its implementation does not translate well on to the existing OP2 active library workflow. While researching more similar applications of the concept, the *easy::JIT* library was discovered.

easy::JIT

easy::JIT [34] is a library created by Juan Manuel Martinez Caamaño and Serge Guelton of *Quarkslab* [1]. It targets C++ code, and utilises `clang` [6] 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 (IR), where other C compilers like `gcc` cannot.

easy::JIT utilises a code generation stage at run-time, and a cache of code to ensure this does not need to be done on every iteration. Performing code generation at run-time means it also needs to be an optimised process, as it will be contributing to the execution time of the application. This has not previously been a consideration for OP2, so the code generation stage is not so easy to integrate into the JIT compilation process.

The OP2 implementation discussed in this report will generate all of the code prior to run-time, as the code generation stage of OP2 is not an optimised process.

Applications developers for OP2 are not currently limited to only LLVM-based C compilers like `clang`, and can use any C compiler they prefer. A translator using LLVM Intermediate Representation to replace the current Python and MatLab source-to-source translation scripts is currently in development [21], which would

bring it more in line with Java, having an original source, an intermediate representation, and then native code after the second compilation stage. It is not the version of OP2 that is the focus of this project however.

The implementation completed for this report is building on the compiler agnostic OP2 implementation, and therefore will not utilise LLVM IR. The next section details the Specification for the implementation, including the project requirements and a system model.

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 pre-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 Pre-Existing System

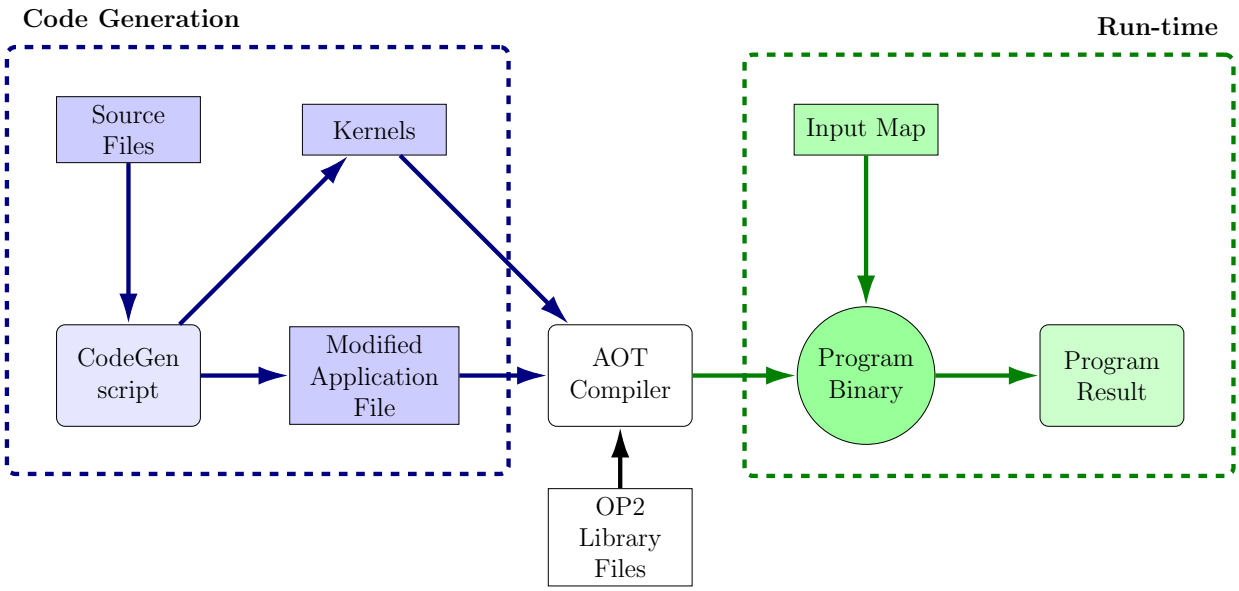


Figure 10: Pre-existing OP2 System Diagram

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 a Master Application file. The Master Application file is a normal C program, which makes OP2 API calls to define of sets, maps, and constants, as well as for initialising and cleaning up the OP2 execution with `op_init()` and `op_exit()`. It describes the application structure, and when parallel loops should be executed.

The input Source Files can optionally contain additional 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 loop body.

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 GPU 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 pre-existing system is able to generate optimised code for GPUs from the high level application code, and apply optimisations ahead of time. It is not, however, able to optimise based on the inputs at all, as they are only known at run-time, 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. Each of these other scripts performs the code generation for a specific hardware platform. `op2.py` uses the Python Regular Expressions (RegEx) library: `re` [41] 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.

Information is also gathered during the parse 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 [16], 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 will generate a definition of every parallel loop's execute function in the form of Kernel files. At compile-time the definitions given in these Kernel files will be linked to the extern declaration in the Modified Application File by the linker.

The requirements for the code generation script that will be created as part of this project, that will produce kernels containing CUDA code with JIT compilation, will be 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 (AOT) 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 [32] 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 [11, 10] without errors.

When the resulting executable which 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 [24, p3].

Lastly, the above requirements must be met for both Array-of-Structs and Struct -of-Arrays data layouts, especially when automatic SoA conversion is enabled, as this

alters the generated code.

3.2.2 Run-time Assertions

The application's input will be a large number of data points forming an unstructured mesh which it will operate over. 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 optimisation, 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 so they are seen as a literal value rather than a memory read 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 of the need to store constant values in memory being removed, retrieval time from memory when a constant value is required has been eliminated. The literal value is immediately available. Other possible optimisations will be discussed in Section 6.1 (Future Work).

The overall goal of this project is to investigate whether this technique does provide any performance benefit, however any performance increase that incurs unacceptable deviation from the expected result is not a useful benefit. Therefore, the addition of defined constants should not reduce the accuracy of the result outside tolerance. It is not expected that it will.

3.2.3 OP2 Library

Outside the translation 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 its runtime needs to be benchmarked to determine if there is performance gain.

The runtime will be compared against the same application generated for graphics cards without JIT compilation, to 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

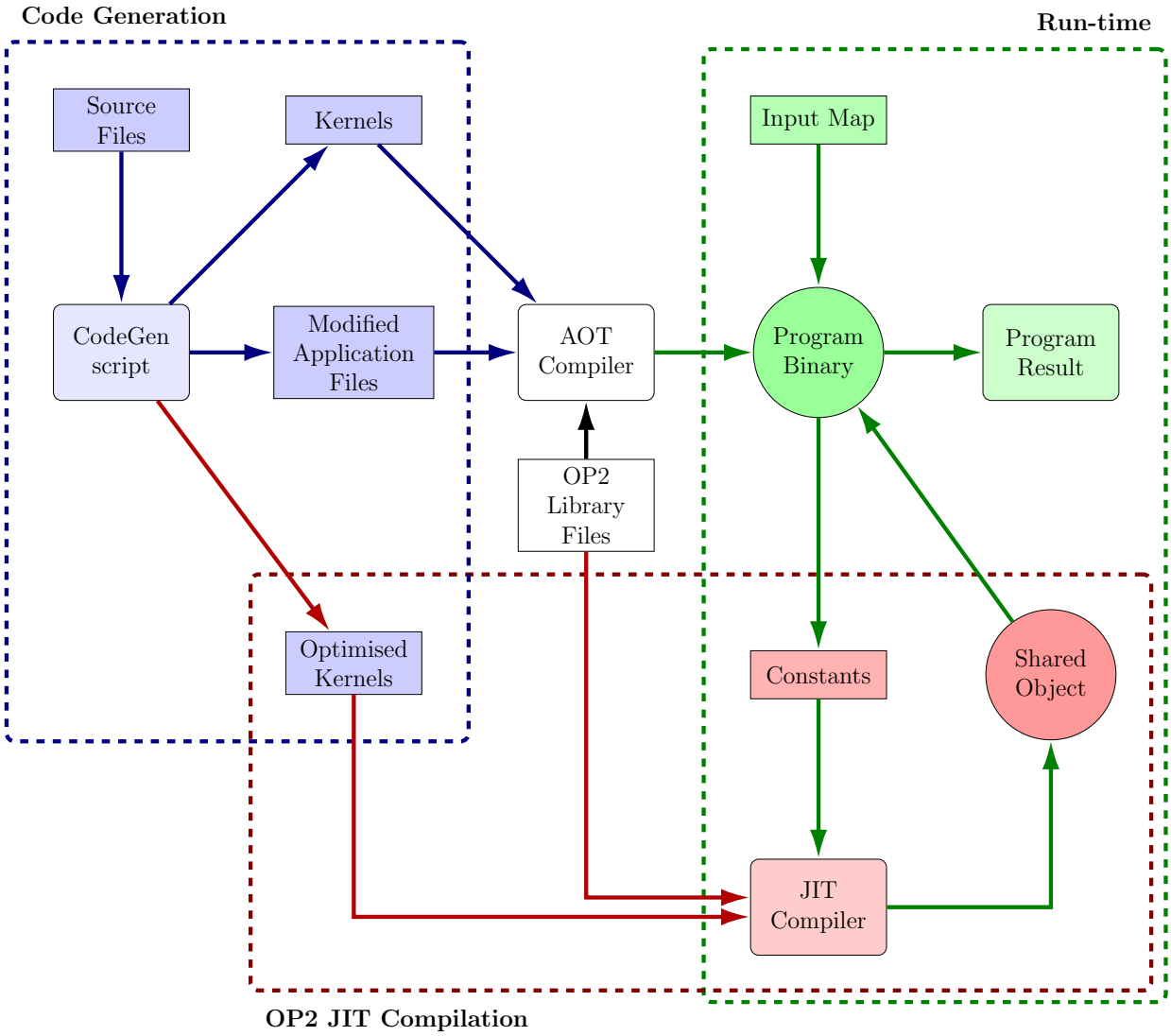


Figure 11: OP2 System Diagram with JIT Addition

Figure 11 describes the new workflow of the OP2 library, with Just-In-Time compilation (denoted with a dotted red box). 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 will 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 compilation will take place during the execution of the binary, and will therefore make up part of the program's execution duration. It will need 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.

The result of JIT compilation will be a Shared Object file, otherwise known as a Dynamically Loaded Library (DLL) file, with a standardised name. The program can then load this Shared Object and utilise its exported functions, which will be the recompiled versions of each parallel loop. As black boxes the two are equivalent (i.e. they have the same inputs and outputs), however theoretically the recompiled versions are faster to execute than the original versions.

Ahead-Of-Time Kernels

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 will 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 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. Constant values should still be copied to device memory, but only if the re-compiled kernels will not be used, and copying the values is necessary.

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)
```

[23, 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 the implementation for this project, it will need 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 a header file, at which 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 that were 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[18] 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

The branch for this project, named `feature/jit`, was initially branched from `feature/lazy-execution` on 13th November 2019. It was lagging behind the `master` branch somewhat, so synchronisation using `rebase` was required before any other changes could be made.

Its parent branch was created for developing a system to execute parallel loops when resulting values are required, rather than when they are called, using an internal library function that takes a descriptor of the loop function, including a void pointer to another function for actually executing it which it will call at a time it deems appropriate:

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

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

Currently this function executes the queued loop immediately, but it is part of ongoing work. Lazy execution will not be the focus of this project, however this process for invoking parallel loops will be utilised in this implementation, so that future efforts towards lazy execution can be continued on top of JIT compilation.

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 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 only modification made is adding a call to the new code generator script 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 for each parallel loop, referred to as **kernel files**. 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, an excerpt of which is shown in Figure 12.

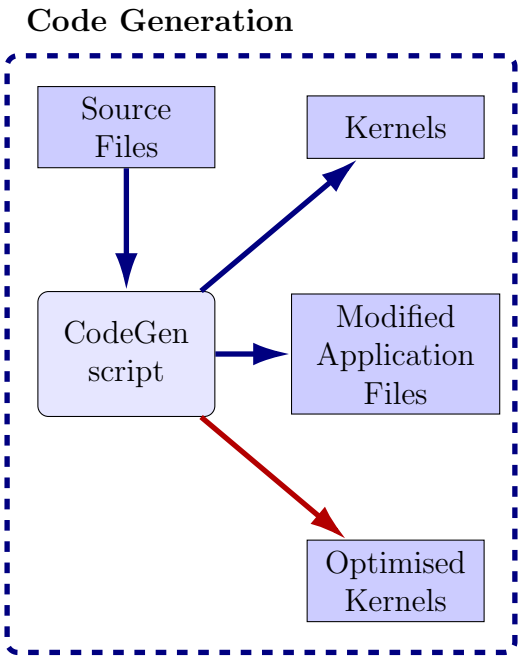


Figure 12: Subsection of System Diagram

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 will be collated into a single file by the compiler.

4.2.2 Execution Setup

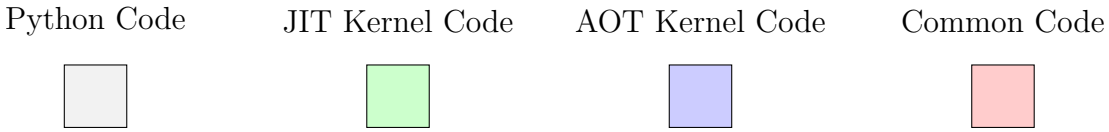
The first action performed by the code generation script when it is invoked is to check across all kernels for the Struct-of-Arrays data layout, or if all are using the default Array-of-Structs. If any do use SoA, a flag named `any_soa` becomes a non-zero value, so will evaluate as `True` in a conditional.

Then, a folder `cuda/` is created if it does not already exist, and the script will iterate over each loop. Both the Ahead-Of-Time (AOT) Kernel file, and the Just-In-Time (JIT) Kernel file are generated simultaneously, as described in Section 4.2.3.

4.2.3 Kernel Files

The code generator outputs two kernel files containing C source code 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 from the translation script will also always be in a frame filled grey, while generated C code from the output files will be in a 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.



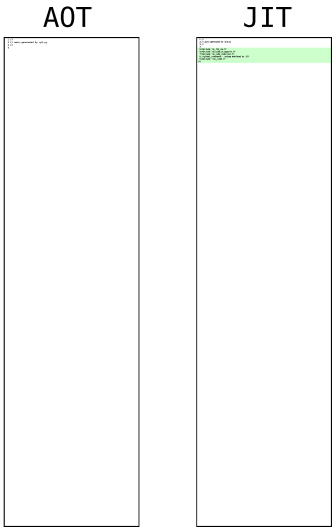
Figures 13-18 show the progression of the two kernel files for a typical parallel loop during the execution of the code generation script (starting from empty files). They are provided only for the purpose of highlighting the relevant sections of each file, and the generated code in the figures is not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

1. JIT includes

The first piece of C code generated by the Python script is simply a number of include directives referencing the OP2 library files. These are only needed by JIT compiled kernels since they will be processed individually by the compiler, in separate processes, so each kernel requires a reference to the OP2 library files.

AOT kernels do not require them, as the Central Kernels File will contain these same `#include` statements:

Figure 13: Kernel Files with JIT includes




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

generated by op2_gen_cuda_jit.py [167-169]

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]

The Python code for generating these statements makes use of the `code()` and `comm()` helper functions from the top of the script, 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

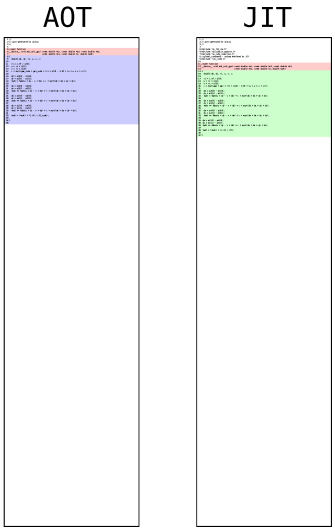
The User Function is the operation specified by the user to be carried out on each iteration of the loop. This function will run on the device (GPU) on many threads simultaneously, performing an action at least once for each set item.

The User Function is given the `__device__` function descriptor, so that it will be compiled for execution on a GPU device, and 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 op2_gen_cuda_jit.py [283]

Figure 14: Kernel Files with 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 when compiling C normally.

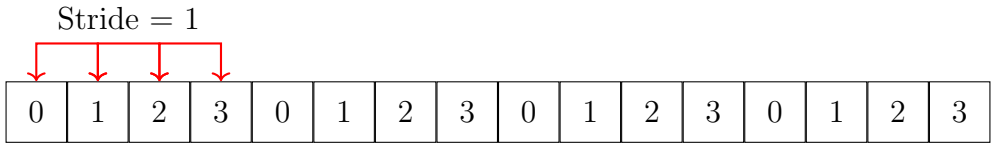
Data Layout If the flag for automatic Struct-of-Arrays data layout transformation is not enabled, the function body will remain largely the same as defined by the application programmer. However, if it is enabled, there are modifications that

need to be made to the function body to achieve this.

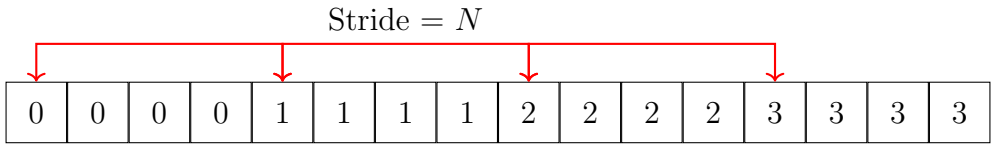
The code for making this transformation is pulled from the pre-existing AOT CUDA code generation script:

`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']
|  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. References to 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. Also, CUDA does not allow variables to be declared both `__constant__`, and given external linkage using `extern` [12, p126], which is how they are handled for the sequential JIT implementation.

The solution to this challenge comes in two parts. For each index `i` of the constant array, a 1 dimensional constant would be defined with the name: `[id]_[i]_OP2CONSTANT`. 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:
|    # Replace all instances with literal int index
|    jit_user_function = re.sub('\\b'+varname+'\[([0-9]+)\]',
                                 varname+'_\g<1>_OP2CONSTANT',
                                 jit_user_function)
```

op2_gen_cuda_jit.py [970-974]

However, if the constant is accessed using any expression other than an integer

literal, this system will run into an issue.

As an example, see the result of processing the following statement, where `c_arr` has a dimension greater than 1, and the index does not match the pattern:

```
int A = c_arr[1 + 2]      ⇒      int A = c_arr_1 + 2_OP2CONSTANT
```

Applying the same replacement here leads to an undefined identifier error at compile-time. In the example above, neither `c_arr_1` or `2_OP2CONSTANT` are identifiers known to the compiler.

To resolve this problem, a constant device array is declared in global scope above the top of the function, with the identifier `[name]_OP2CONSTANT`. Each index of the array will be given the value from 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 performance could be reduced by allocating a new array. If there are no expression accesses, the code will not be generated to handle them. The code below demonstrates how the example from the top of the page will be handled.

```
|__constant__ int c_arr_OP2CONSTANT = { c_arr_1_OP2CONSTANT, ... };
...
|_device__ void [name]_gpu ( [args] )
|{
|  int A = c_arr_OP2CONSTANT[1+2];
...
```

generated by op2_gen_cuda_jit.py [979-994]

The above is a trivial example, and actual application code is unlikely to use an expression involving only literal values. If it were, there would be benefit to implementing constant folding [7] to evaluate the expression at compile-time where possible, and reduce an expression access to a literal value access.

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+'\[([0-9]+)\]',
                               varname+'_\g<1>_OP2CONSTANT',
                               jit_user_function)
|
|    # Replace and count all remaining array accesses
|    jit_user_function, numFound = re.subn('\\b'+varname+'\[',
                                          varname+'_OP2CONSTANT[',
                                          jit_user_function)
|
|    # At least one expression index found
|    if (numFound > 0):
|        if CPP:
|            #Line start
|            codeline = '__constant__ '           +\
|                        consts[nc]['type'][1:-1] +\
|                       ' '+varname               +\
|                       '_OP2CONSTANT'            +\
|                       '['+consts[nc]['dim']+'] = {'
|
|            # Add each constant index to line
|            for i in range(0,int(consts[nc]['dim'])):
|                codeline += varname+'_'+str(i)+'_OP2CONSTANT, '
|
|            # 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 [970-999]

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 explanation as to why in a later section on the Host Function.

3. Kernel Function

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

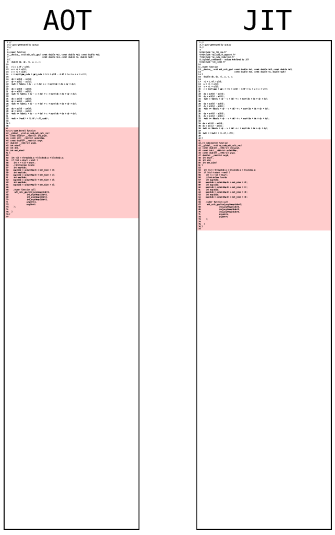
The Kernel Function is the same in both files, and is also executed on the GPU across all the parallel threads. It is declared `__global__` so that it will be executed on the device, but can be called from host (CPU) code:

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

generated by op2_gen_cuda_jit.py [299]

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 Files with Kernel Function



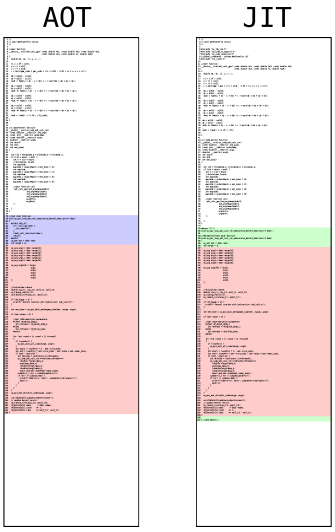
Indirection: If the loop is indirect, and uses values from another map as indices, these values need to be read from the inner map in this function, so that the User Function (generated above) can receive all the data already formatted in the manner it expects to receive it. It is possible that the indirect 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 [23, 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: Kernel Files with 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 [536-561]

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.

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`, otherwise it will be disabled by default. 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 compiled 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 compilation 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 [549-558]

JIT: Contrasting the code generated for a JIT enabled kernel on the next page 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 [16].

```
|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( ... )
```

Instead of:

```
op_par_loop_[name]_execute( ... )
```

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);
```

generated by op2_gen_cuda_jit.py [554]

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 will be used.

Theoretically the JIT version of the user function will complete fractionally quicker than the unoptimised version, and the difference will become noticeable if the number of times it is called is sufficient.

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.

```
|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 [643-657]

These sizes only become available when the function is first called and its 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 unacceptable, as each loop would need to have been called at least once before the compilation could be done, so that all the strides are known.

For this to be possible, each loop would need to execute correctly both before and after JIT compilation in a binary with JIT compilation enabled, and therefore all the input constants would need to be copied to device memory as usual, adding extra duration to the upfront cost of the optimisation.

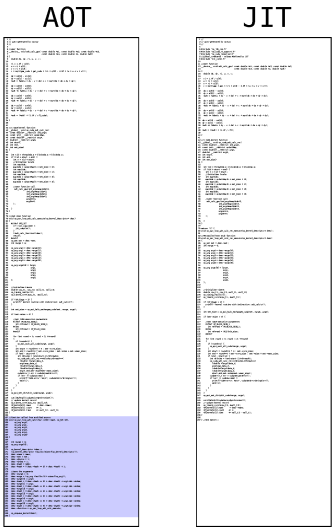
Furthermore, parallel loops may not all be created equally. It is a possibility that a certain loop never gets called, or is only called for the first time half way through an application. In a situation such as that, any benefit that could be gained from JIT compiling would be wasted while waiting for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

Figure 18: Kernel Files with Loop Function



The Application File will be modified by `op2.py` to contain a 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:

```
|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 execute when the loop is called by the Application File, and generate an `op_kernel descriptor` out of the loop. The kernel descriptor is an OP2 data structure that holds the name, arguments, and a pointer to the execution function of the loop. It is passed as an argument to the enqueue function so the loop can be executed via the function pointer at a time decided by the lazy 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 be executed when the corresponding loop is invoked in the Application File, which will have 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 program 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 the Kernel Function is then 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 application programmer's design.

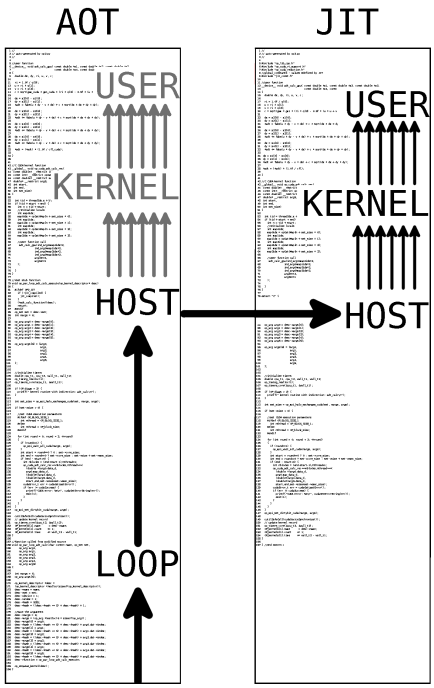


Figure 19: Kernel Flow

The `jit_compile()` function, which will actually perform the 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 will reside 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. The remaining loose ends will be tied up, as it contains the `jit_compile()` function for invoking the run-time compiler, and the new definition for the OP2 API function for declaring constants.

It also contains `#include` statements for each of the AOT kernel files, so that their contents are 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 op2_gen_cuda_jit.py [1020-1025]

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 application 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 [1029-1040]

These declarations are generated by 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 it exports.

Recall from the System Model (Figure 20) that the Shared Object has an arrow back into the Program Binary, which represents this process. This allows the newly compiled loop functions to be found in memory, and used by the executable when required.

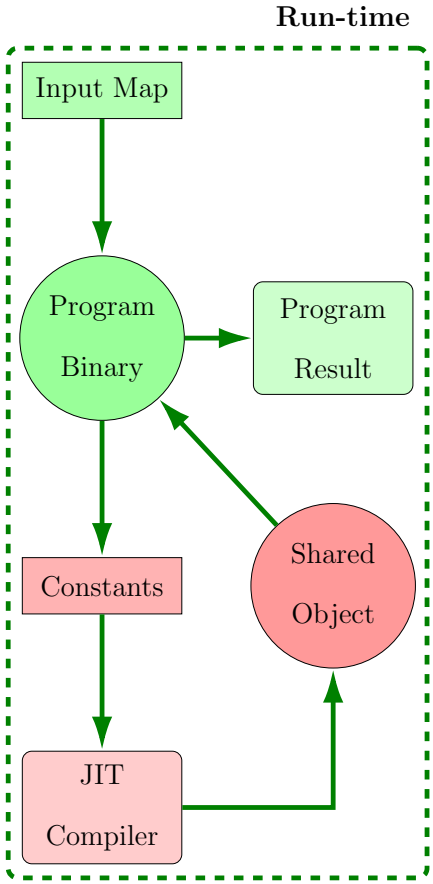


Figure 20: OP2 System Diagram Run-time Sub-section

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 [23, p9]:

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

To ensure backwards compatibility this signature cannot be altered. 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 it is not necessary to include it in both versions.

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 have all been declared the header file defining each value can be generated.

As can be seen in the generated C code on the next page, constants containing more than one value are iterated over, and each element is declared as a single value due to the issues with `extern __constant__` values described in Section 4.2.3 (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 for making their functions available to the binary. To gather the time spent compiling the binary, `jit_compile()` uses the same OP2 library timing functions as the kernels use to record the time spend in each parallel loop. It will be important that the time taken to compile at run-time is known for performance comparisons later.

Above the `jit_compile()` function, in global scope, a function pointer is declared for each parallel loop, and defined `NULL`. 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 [31] 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, such as library install paths, 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 to be 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* [17].

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 it 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 about 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, completing the implementation required it created, and therefore it 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 the parameters that will be passed to the compiler process, and other options. A number of other libraries are still required to build an OP2 binary, as they were before JIT compilation was added (Appendix B - *Getting Started with OP2*). In this section only the recompilation target will be discussed, as it was the new target produced 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 invoked at run-time. As mentioned in the previous section, the result of making this target needs to be 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 `nvcc` 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 itself has not been modified, otherwise the make flag would not function correctly, and an old, JIT enabled version of this file would be used even if the variable is set to `FALSE`.

4.3.1 Optional Functionality

By default, the JIT compilation functionality is enabled in the example Makefile, 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 file with JIT compilation disabled.

The example Makefile is provided in `apps/c/airfoil/airfoil_JIT/dp/` of the GitHub repository. This is the directory used for testing and benchmarking the JIT compilation implementation, using the *airfoil* OP2 application, as described in the next section.

5 Testing

Throughout development, an example application that was previously developed using the OP2 API was used to test code generation, and verify the results. The application is called *airfoil*, and it has been used for validating generated OP2 code before [36], as it makes use of all the key features including having both direct and indirect loops.

airfoil is a computational fluid dynamics application which models the air flow around an aeroplane wing, using unstructured grid to discretise the space. A document detailing the *airfoil* code is available on the OP2 website [22]. Figure 21 is an simple 120 x 60 mesh for *airfoil*, showing the wing shape, and increasing granularity close to the shape. Each quadrilateral represents a cell.

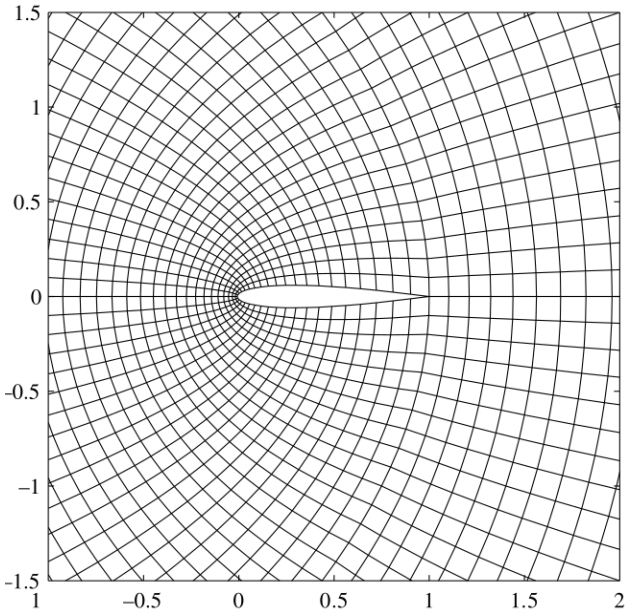


Figure 21: Rendering of an *airfoil* mesh. Diagram from [36]

To ensure that the test cases selected definitely validate the implementation, the requirements set out in the Specification must be revisited. They are summarised in the next Section.

5.1 Requirements

1. Source code files must be produced as the output of the new code generation Python script.

2. The generated source code files must be valid.

   i. C code must compile using `icc` without errors;

   ii. CUDA code must compile with `nvcc` without errors.

3. The compiled executable must invoke a re-compilation stage, if the feature is enabled.

   i. Re-compilation must also complete without error;

   ii. The binary must execute code that has been compiled during its execution.

4. Constant values from the input must be available to the User Function.

   i. If JIT is enabled, they must be turned into `#define` directives;

   ii. Otherwise they must be copied to device memory.

5. The compiled executable must produce a result within some tolerance of the expected result.

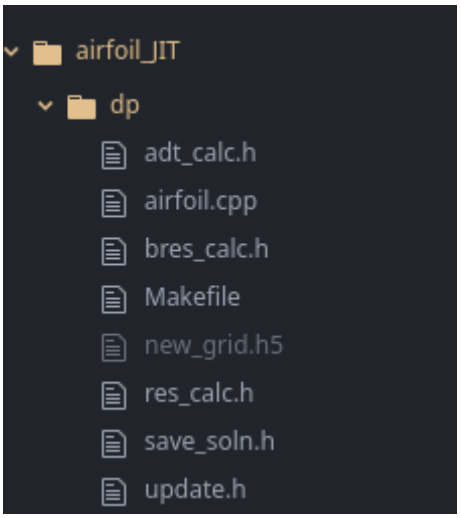
6. The OP2 API must not be modified.

Section 5.3 is the final results of the test plan used throughout the project whenever a work done needed to be validated. Each test ensures that a requirement from the above list has been met, and also includes the date when it first passed.

Requirement 6 can be trivially accepted when testing, since the implementation did not modify the OP2 API. There is no test for this requirement.

5.2 Initial State

Figure 22: *airfoil* folder initial state



The initial state for testing is a folder containing the source files for *airfoil*, as listed in Figure 22. The Application File is `airfoil.cpp`, and the five header files each contain a User Function for the parallel loop with the same name.

`new_grid.h5` is the name of the input data file, formatted in the Heterogeneous Data Format (HDF5) [26]. This file can be obtained from the OP2 website [38], and must be converted from `.txt` to `.h5` using the `convert_mesh` tool.

OP2 already provides a standard set of functions for performing file I/O on an HDF5 file, which *airfoil* uses. The contents of the file can be viewed using the `hdump` utility, which comes with an HDF5 installation.

5.3 Test Plan & Results

1. "Source code files must be produced as the output of the new code generation Python script."

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 GPU code generation script is used.

The environmental variable `$OP2_INSTALL_PATH` can be assumed to hold to full path to the `op2/` folder in the top level of the OP2 repository. Since the translation script is in a sub-directory of `translator/`, which is also a top-level folder, the path to the Python translator script will be as shown below.

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

After running this command in the *airfoil* directory, the expected outcome is that a new file: `airfoil_op.cpp` is created, as well as a new directory named `cuda/`, which will contain with eleven CUDA source code files it: two kernel files for each of the five parallel loops, and a single Central Kernels File, named `airfoil_kernels.cu`.

This test is considered a pass if these files exist, as their contents will be validated as correct if the following tests pass. 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` branch.

If the environmental variable `OP_AUTO_SOA` is set to one, the code will be generated with transformations to automatically use the Struct-of-Arrays data structures, overwriting any generated source files that exist already. To confirm the functionality is working with both, the generated files need to be deleted, and the script invoked again once the environmental variable has been set.

Figure 23: *airfoil* folder after Code Generation

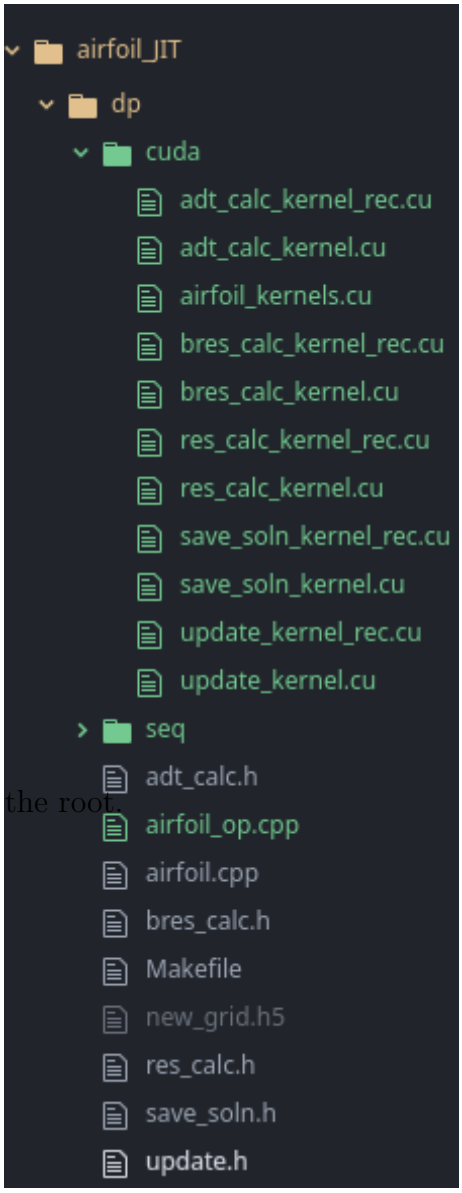


Figure 23 shows the folder after a passing test case of running the OP2 translation script. The folder appears identical whether automatic

Struct-of-Arrays transformation is enabled or not, which is the expected outcome, so the Figure has not been duplicated.

When compared to Figure 22, all the additional files have been generated by the code generation script. As expected, there are two kernel files for each parallel loop, and a single Central Kernels File in the `cuda/` directory, and a Modified Application file in the root.

Test **PASSED**. 03/12/2019

2. "The generated source code files must be valid."

The validity of the generated source code files will be confirmed by the initial compilation of the binary completing without error, both when JIT compilation is enabled and disabled. Recall that even with JIT compilation enabled, the AOT compiler is still required to produce an initial binary. If any errors are produced by the compiler, the code generated is not valid, and therefore is of no further use.

In the `airfoil_JIT` folder, compilation is done using the Makefile, and for the initial compilation of the binary, the target is `airfoil_cuda`. The Makefile produces a binary with JIT compilation enabled by default, so the command to compile it will be:

```
make airfoil_cuda
```

Whereas, to produce a binary that will execute only Ahead-Of-Time compiled code, the command will be:

```
make airfoil_cuda JIT=FALSE
```

The resulting 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$OP2_INSTALL_PATH/c/include
   -I$HDF5_INSTALL_PATH/include -Icuda -I.
   -c -o cuda/airfoil_kernels_cu.o cuda/airfoil_kernels.cu
```

The inclusion of `-DOP2_JIT` depends on which of the two above commands was called. Both commands will need to be tested for errors when the code base has been generated both with and without automatic Struct-of-Arrays transformations, giving four test cases. The generated code will also need to be manually inspected to ensure that transformations have been made when `OP_AUTO_SOA` is enabled.

Figure 24: *airfoil* folder after Ahead-Of-Time compilation

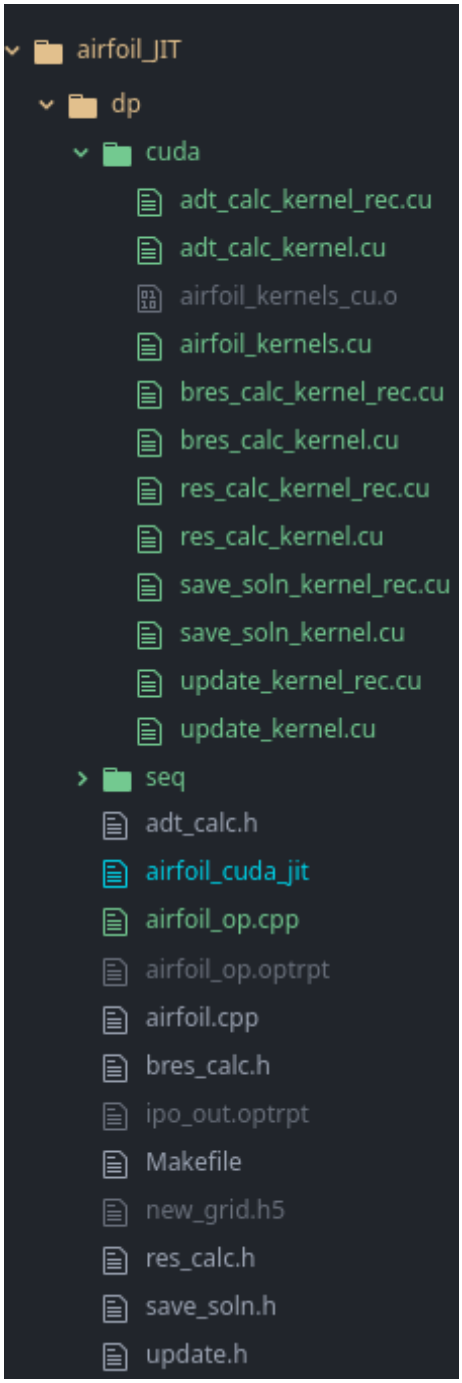


Figure 24 shows the folder after the make command has been successfully executed. As before, the folder will appear almost identical for all four test cases, so the Figure has not been duplicated.

In the figure the first new file that can be seen is a single object file in the `cuda/` folder, which is the result of compiling the Central Kernels File and all the kernels that are not marked `_rec` into a single object binary.

In the parent directory the executable `airfoil_cuda_jit` has been generated, which statically links the above object file, so does not require access to it at run-time. If JIT compilation is not enabled, the binary is just called `airfoil_cuda`.

Lastly, a number of optimisation reports have been generated by the Intel C compiler.

The test results and dates for the four test cases are shown in a matrix below.

JIT	`OP_AUTO_SOA=0`	`OP_AUTO_SOA=1`
TRUE	**PASSED**. 16/01/2020	**PASSED**. 17/01/2020
FALSE	**PASSED**. 16/01/2020	**PASSED**. 17/01/2020

3. "The compiled executable must invoke a re-compilation stage, if the feature is enabled."

For this test to be considered a pass, a compiler process must be started during the execution of the binary, and must complete without producing errors. 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 executable printing the compilation duration to the console output like the example below confirms the compilation has completed, and success can be confirmed by checking the compiler log for errors. If none are found, requirement **3i** has been met: **"Re-compilation must also complete without error"**.

Figure 25: Example success output

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

It should be the case that these lines are **not** printed if the executable was compiled with JIT compilation disabled, otherwise the part of the requirement that states **"..., if the feature is enabled"** has been violated.

In order to determine whether sub-requirement **3ii** has been met: **"The binary must execute code that has been compiled during its execution"**, one of the Kernel files that will be compiled at runtime is manually edited to contain a print statement, or some other identifier to confirm which version is being executed: the run-time compiled or the original.

As with the second requirement there are 4 test cases. It is possible that the state of `OP_AUTO_SOA` could interfere, so both enabled and disabled will need to be tested for both possible states for JIT compilation.

Figure 26: *airfoil* folder after Just-In-Time Compilation

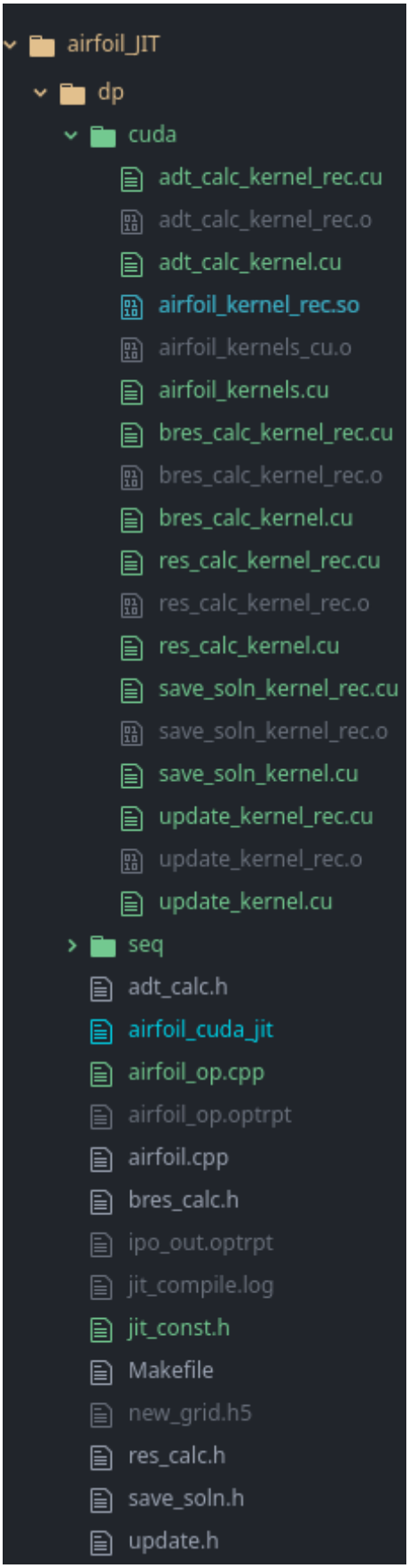


Figure 26 shows the *airfoil* folder after the JIT compilation stage has completed successfully. In the Figure, there is now an object file for each of the parallel loops, as well as a new shared object in the `cuda/` folder, which will have been loaded by the running executable. Some other files have also been generated, including the compilation log file `jit_compile.log`, and the constants header file `jit_const.h` which is part of the next requirement.

The JIT compilation log file does not contain any errors, and the expected files have been generated, so the main part of the requirement has been met.

Manually adding a print statement to only the re-compiled kernels confirms that the correct kernels are being executed, and JIT compilation is enabled, the executable is utilising functions that have been compiled as part of it's execution.

JIT	`OP_AUTO_SOA=0`	`OP_AUTO_SOA=1`
TRUE	**PASSED.** 22/01/2020	**PASSED.** 22/01/2020
FALSE	**PASSED.** 22/01/2020	**PASSED.** 22/01/2020

4. "Constant values from the input must be available to the User Function."

This test simply requires that the User Function is able to access the values of input constants when they are executing on the GPU. For Kernels compiled with JIT compilation disabled, this requires that they are copied to device memory, and for JIT enabled Kernels they must be defined literals.

The *airfoil* program includes a number of input constants, where six have a dimension of 1, and one has a dimension of 4 named `qinf` - all holding values of the type `double`. It never exercises the functionality of accessing `qinf` using an expression however, so to test that this functionality works correctly, additional code needs to be added inside the User Function.

This can be solved at the same time as testing the other constants by adding to the code generation script, such that inside the User Function the values of constants are printed. This should include a loop over `qinf` such that it is accessed using an index variable, and with literal values.

```
|IF('blockIdx.x == 0 && threadIdx.x == 0')
|for nc in range (0,len(consts)):
|  comm(consts[nc]['name'])
|  if consts[nc]['dim']==1:
|  code('printf("'+name+'-'+consts[nc]['name']+
        ': %1.17e\\n",'+ consts[nc]['name']+');')
|  else:
|    FOR('i','0',consts[nc]['dim'])
|    code('printf("'+name+'-'+consts[nc]['name']+
          '[%d]: %1.17e\\n", i,'+consts[nc]['name']+'[i]);')
|    ENDFOR()
|    for i in range (0,int(consts[nc]['dim'])):
|      code('printf("'+name+'-'+consts[nc]['name']+
            '['+str(i)+']: %1.17e\\n",'+ consts[nc]['name']+
            '['+str(i)+']);')
|ENDIF()
```

The above code is left in the translator script, at lines [259-281], but has been commented out as printing this much to the terminal is very detrimental to performance.

It should not be included for benchmarking or production code generation.

For *airfoil*, the code added to the end of the JIT enabled `save_soln` User Function is:

```
|  if (blockIdx.x == 0 && threadIdx.x == 0) {
|    //gam
|    printf("save_soln-gam: %1.17e\n",gam);
|    //gm1
|    printf("save_soln-gm1: %1.17e\n",gm1);
|    //cfl
|    printf("save_soln-cfl: %1.17e\n",cfl);
|    //eps
|    printf("save_soln-eps: %1.17e\n",eps);
|    //mach
|    printf("save_soln-mach: %1.17e\n",mach);
|    //alpha
|    printf("save_soln-alpha: %1.17e\n",alpha);
|    //qinf
|    for (int i = 0; i < 4; ++i)
|    {
|      printf("save_soln-qinf_OP2CONSTANT[%d]: %1.17e\n", i,
               qinf_OP2CONSTANT[i]);
|    }
|    printf("save_soln-qinf_0_OP2CONSTANT: %1.17e\n",
             qinf_0_OP2CONSTANT);
|    printf("save_soln-qinf_1_OP2CONSTANT: %1.17e\n",
             qinf_1_OP2CONSTANT);
|    printf("save_soln-qinf_2_OP2CONSTANT: %1.17e\n",
             qinf_2_OP2CONSTANT);
|    printf("save_soln-qinf_3_OP2CONSTANT: %1.17e\n",
             qinf_3_OP2CONSTANT);
|  }
```

When the above lines are included, the expected output for each loop is that each one dimensional constant will be printed along with the function it is being used in. Then, the multi value constants will be printed twice: once using a variable `i` to index it, and again using literal values.

The generated code for an AOT kernel is similar, but with modified references to the constants, since constant references are managed using a string replacement on all places where they appear in the User Function.

When the `save_soln` loop is executed, the terminal output with JIT compilation is enabled will be the following if the values are available and correct:

```
save_soln-gam: 1.39999997615814209e+00
save_soln-gm1: 3.99999976158142090e-01
save_soln-cfl: 8.99999976158142090e-01
save_soln-eps: 5.00000007450580597e-02
save_soln-mach: 4.00000005960464478e-01
save_soln-alpha: 5.23598775598298830e-02
save_soln-qinf_OP2CONSTANT[0]: 1.00000000000000000e+00
save_soln-qinf_OP2CONSTANT[1]: 4.73286385670476317e-01
save_soln-qinf_OP2CONSTANT[2]: 0.00000000000000000e+00
save_soln-qinf_OP2CONSTANT[3]: 2.61200015044213218e+00
save_soln-qinf_0_OP2CONSTANT: 1.00000000000000000e+00
save_soln-qinf_1_OP2CONSTANT: 4.73286385670476317e-01
save_soln-qinf_2_OP2CONSTANT: 0.00000000000000000e+00
save_soln-qinf_3_OP2CONSTANT: 2.61200015044213218e+00
```

Furthermore, when an *airfoil* binary with JIT compilation enabled is executed, the `jit_const.h` file should contain the following statements. Values correspond with those above.

```
|#define gam 1.39999997615814209e+00
|#define gm1 3.99999976158142090e-01
|#define cfl 8.99999976158142090e-01
|#define eps 5.00000007450580597e-02
|#define mach 4.00000005960464478e-01
|#define alpha 5.23598775598298830e-02
|#define qinf_0_OP2CONSTANT 1.00000000000000000e+00
|#define qinf_1_OP2CONSTANT 4.73286385670476317e-01
|#define qinf_2_OP2CONSTANT 0.00000000000000000e+00
|#define qinf_3_OP2CONSTANT 2.61200015044213218e+00
```

The test results and dates for the four test cases are shown in a matrix below.

JIT	`OP_AUTO_SOA=0`	`OP_AUTO_SOA=1`
TRUE	**PASSED**. 21/01/2020	**PASSED**. 21/01/2020
FALSE	**PASSED**. 21/01/2020	**PASSED**. 21/01/2020

5. "The compiled executable must produce a result within some tolerance of the expected result."

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

Expected Result The *airfoil* OP2 application prints the value of the `rms` (root mean square) variable every 100 iterations. According to the documentation, the first 1000 iterations for double precision should be exactly:

		`rms`
Iterations	100	5.02186×10^{-4}
	200	3.41746×10^{-4}
	300	2.63430×10^{-4}
	400	2.16288×10^{-4}
	500	1.84659×10^{-4}
	600	1.60866×10^{-4}
	700	1.42253×10^{-4}
	800	1.27627×10^{-4}
	900	1.15810×10^{-4}
	1000	1.06011×10^{-4}

Table 1: Expected values of rms

The application code also includes a test of the result after 1000 iterations, which compares against the expected outcome and prints the calculated percentage difference using the equation:

$$\%diff = \left|(100 \times \frac{rms}{0.0001060114637578}) - 100\right|$$

A difference of less than 0.00001% is considered within tolerance due to the potential for minor floating point errors.

Actual Results

The tables below show the results output by the compiled binary every 100 iterations.

Table 2: Console Output when JIT compilation is Enabled

		`OP_AUTO_SOA=0`	`OP_AUTO_SOA=1`
Iterations	100	5.02186×10^{-4}	5.02186×10^{-4}
	200	3.41746×10^{-4}	3.41746×10^{-4}
	300	2.63430×10^{-4}	2.63430×10^{-4}
	400	2.16288×10^{-4}	2.16288×10^{-4}
	500	1.84659×10^{-4}	1.84659×10^{-4}
	600	1.60866×10^{-4}	1.60866×10^{-4}
	700	1.42253×10^{-4}	1.42253×10^{-4}
	800	1.27627×10^{-4}	1.27627×10^{-4}
	900	1.15810×10^{-4}	1.15810×10^{-4}
	1000	1.06011×10^{-4}	1.06011×10^{-4}
Accuracy		$2.484679129111100 \times 10^{-11}\%$	$2.489120021209601 \times 10^{-11}\%$

Table 3: Console Output when JIT compilation is Disabled

		`OP_AUTO_SOA=0`	`OP_AUTO_SOA=1`
Iterations	100	5.02186×10^{-4}	5.02186×10^{-4}
	200	3.41746×10^{-4}	3.41746×10^{-4}
	300	2.63430×10^{-4}	2.63430×10^{-4}
	400	2.16288×10^{-4}	2.16288×10^{-4}
	500	1.84659×10^{-4}	1.84659×10^{-4}
	600	1.60866×10^{-4}	1.60866×10^{-4}
	700	1.42253×10^{-4}	1.42253×10^{-4}
	800	1.27627×10^{-4}	1.27627×10^{-4}
	900	1.15810×10^{-4}	1.15810×10^{-4}
	1000	1.06011×10^{-4}	1.06011×10^{-4}
Accuracy		$2.486899575160351 \times 10^{-11}\%$	$2.493560913308102 \times 10^{-11}\%$

All of these results are well within tolerance. The requirement has been met.

5.4 Benchmarking

The functionality has now been confirmed to work as intended, and the technical requirements of the project have all been met: new code is able to be generated by the OP2 translator script, which can be compiled into a binary that will execute code it has compiled itself as part of it's execution. What remains is to benchmark the run-time of the application with JIT compilation enabled and disabled, and determine if there is any performance gain.

The following results should be considered a benchmark of the Constant Definition optimisation, rather than JIT compilation as a whole, as further assertions at run-time are possible and could make even more use of the compiler being aware of the input data.

5.4.1 Hardware

Testing was done on a personal computer with an NVIDIA GeForce MX250 Graphics Card [13] - 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 for the runtime of the *airfoil* application. Using a personal computer system may result in noisy data, for example from the Operating System scheduling other tasks.

In order to gather better data, access to a HPC cluster located in Cambridge, part of the Cambridge Service for Data-Driven Discovery (CSD3) [4], was approved - with the caveat that workloads for this project would be placed in a low priority queue.

The supercomputer named *Wilkes2* was used, which is the largest GPU enabled supercomputer for academic research in the UK. *Wilkes2* has 90 nodes, each with the specifications in Table 4 [5].

Table 4: *Wilkes2* hardware specification

CPU	1 x	12-core Intel Xeon E5-2650 v4 2.2GHz	[9]
RAM		96GB	
GPU	4 x	NVidia P100 16GB	[14]

The translator currently only generates code for a single graphics card, so only one of the four will be used. A possible extension would be to include MPI and divide the workload across multiple GPUs.

5.4.2 Benchmarking Strategy

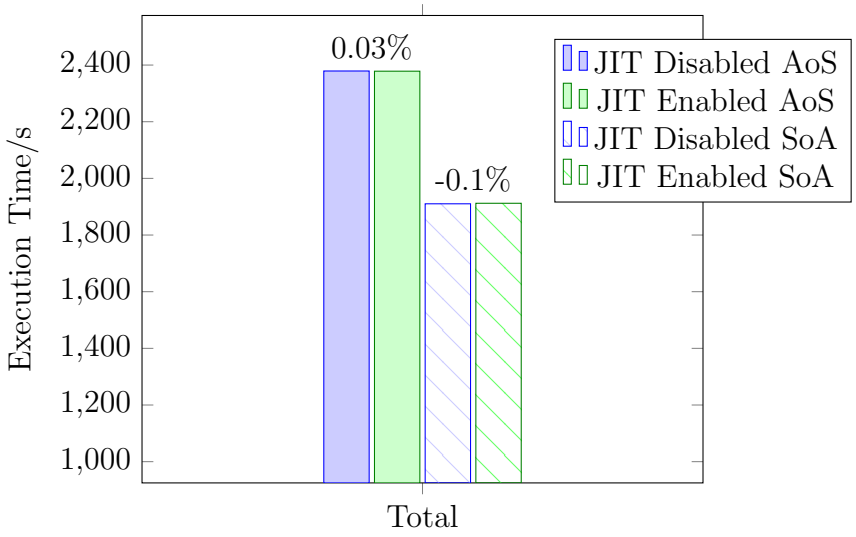
The *airfoil* program was also used for benchmarking, as it is reasonably representative of production applications. The input mesh remains the same size as in the previous Section where it was used for testing functionality, with 721,801 nodes, but the number of time step iterations was upped from 1000 to 1,000,000 to make any differences in run-time more noticeable. OP2's internal timing functions 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 discussed in Section 3 (Just-In-Time Compilation), the time taken for the invocation of the compiler at run-time to complete is also recorded, and will be included in the data. It is a one-time cost at the start of execution, but still needs to be considered.

5.4.3 Results

The graph in Figure 27 shows the mean total run-time of the four different configurations, averaged from five executions, in an attempt to further reduce any noise from factors other than those being tested. The percentage speed-up from the original version (blue) to JIT compiled (green) is shown above each pair of bars.

Figure 27: Total Execution Time



The speed-up percentages above the bars were calculated using the formula below:

$$\%speedup = \frac{\text{Initial Time} - \text{New Time}}{\text{Initial Time}} \times 100\%$$

Therefore a positive value indicates that the JIT compiled version completed faster, while a negative value indicates it was outperformed by the original.

Analysis

In Figure 27, the speed-up for Array-of-Structs data layout is 0.03%, which is only a very small improvement coming to about 1 second saved out of nearly 40 minutes. However the setup cost of run-time compilation is an average of 4.13 seconds, compared to essentially zero seconds to copy constants to device memory (0.0001s for all *airfoil* constants). This duration does not vary significantly when

either the size of the mesh or the number of iterations is increased.

If the re-compilation time is excluded, the speed-up is 0.2% for the actual execution of the application. From this, and the fact that compilation time is O(1) for input size, it follows that the speed-up could increase linearly with problem size, albeit at a shallow gradient. There will always be a constant initial cost, but every single iteration will complete fractionally faster. More iterations means more time saved.

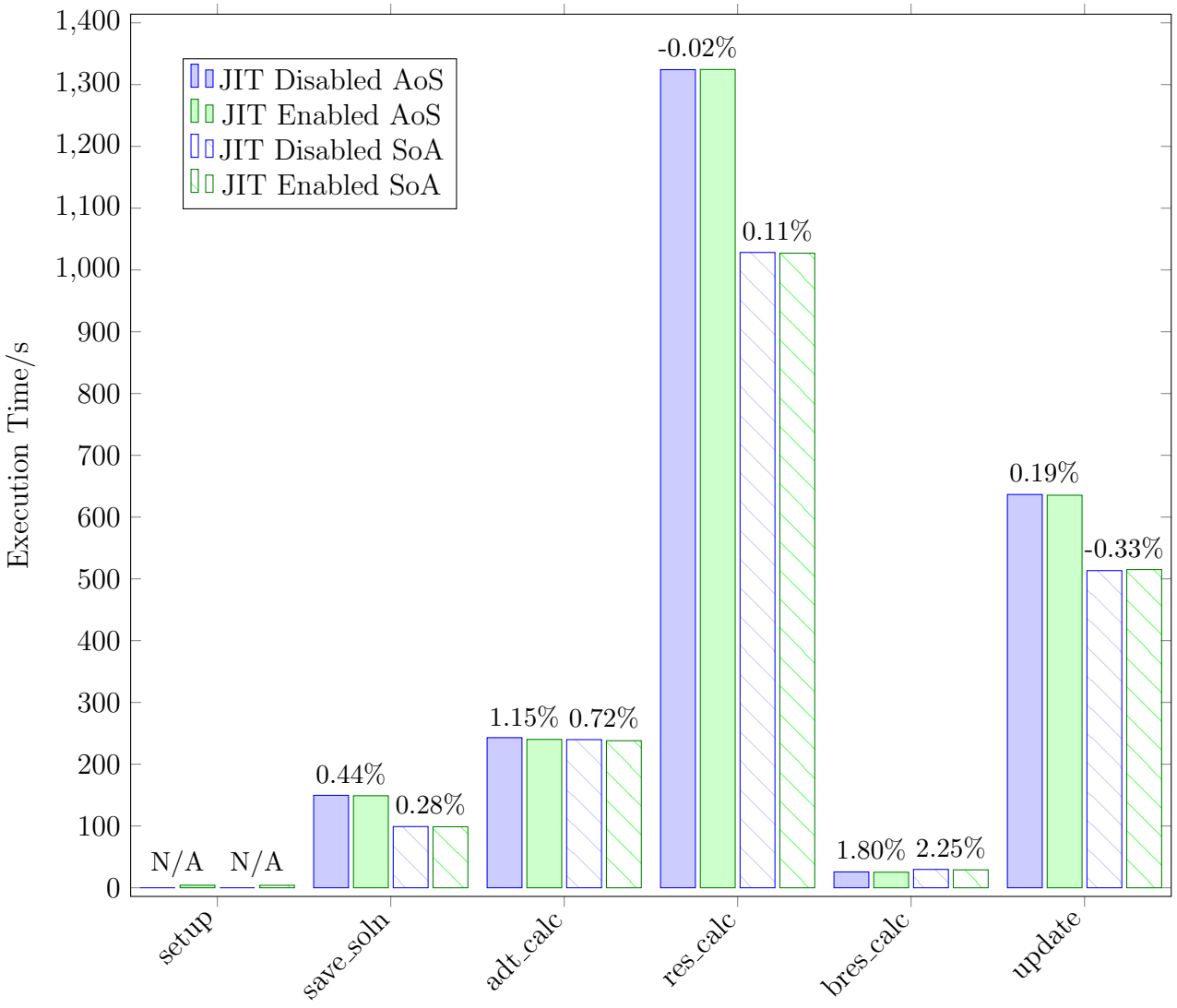
Unfortunately, when `OP_AUTO_SOA=1` the object difference is that the executable with JIT compilation enabled is 0.1% slower. As before, it should be considered that JIT compilation incurs a larger one-time cost at the start, and comparing just the application execution the runtime is 0.14% faster. It would seem there is benefit possible for the SOA enabled build, but the problem size needs to be even greater for the per-iteration time reduction to outweigh the upfront cost, and the total execution time to be reduced.

5.4.4 Results by Parallel Loop

The runtime can be broken down into how long the application spent executing each of the parallel loops, and what speed-up each loop was able to see when applying JIT compilation. In the chart on the next page (Figure 28), the speed-up of each parallel loop is plotted. Once again, a percentage is given above each pair of bars, which is the speed-up of the JIT compiled version compared to the version with JIT compilation disabled.

As in Figure 27, a positive value indicates that the JIT compilation enabled version completed faster, while a negative percentage means the version without JIT compilation had the shorter runtime.

Figure 28: Execution Time by Function



The speed-up of the setup stage is marked Not Applicable (instead of negative four million percent) since the time taken to copy constants to device memory for *airfoil* is essentially zero seconds.

Analysis

As with most HPC applications, different loops in *airfoil* take up different proportions of the runtime. In Figure 28 it is clear that *res_calc* dominates the runtime, and is also the least able to benefit from the Constant Definition optimisation made in the JIT compiled code, since it was actually marginally slower for AoS. Looking into the

source code, the loop body only uses the input constants three times: `gm1` twice in quick succession, so would likely still be cached in the AOT compiled version; then `eps` once a few lines later.

Comparing this to *bres_calc*, which was the best affected by Constant Definition, the pattern holds as this function makes sixteen references to `qinf`, as well as using `gm1` twice. Although this could not be tested in the time available, it would seem logical that if the function that dominates the runtime is also making heavy use of input constants, the overall benefit to the application could be greater.

5.4.5 Results Conclusion

What the results demonstrate most of all, is that there is definitely potential in Just in Time compilation. At one million time steps, the optimisation of Constant Definition was able to approximately make up the cost of re-compilation through fractional reduction in the time taken to perform every iteration, and increasing the problem size would only improve the speed-up.

Since the assertion being made is only that values declared constant will remain constant, the time available for optimisation is only the time taken to read constant values from memory, which will not usually make up a significant proportion of the run-time.

If more optimisations are implemented, which make further use of the inputs being known, and the per-iteration speed-up is increased, then the required problem size to see benefit will shrink, and the technique becomes even more valuable.

Even without further optimisations, a small improvement to a very large solver application, which might be executing many millions of time-step iterations, can quickly outweigh the relatively tiny one-time cost of recompilation, and start to make an improvement to the overall runtime.

6 Evaluation

This project was intended as an investigation, and therefore it can certainly be considered successful. The potential for optimisation through JIT compilation has been proved, via the demonstration that the optimisation of defining constants as literal values is able to re-coup the cost of run-time compilation, if the problem size is sufficiently large.

Since the space for optimisation by this technique is relatively small, the problem size does have to be very large to see benefit. However HPC applications do tend to be very large, hence why they are not run on normal computers. This is not a significant restriction.

It is very promising that this optimisation is able to provide any benefit at all, and as a result of the contributions made to the OP2 Framework while completing this project, important groundwork has been laid for future contributors to build on top of. Further optimisations and run-time assertions can be implemented as extensions to this implementation, which might achieve greater speedup at run-time, especially for smaller problem sizes.

6.1 Future Work

6.1.1 Run-Time Assertions

As previously mentioned, it seems necessary for more assertions to be made at run-time in order to produce more effective speed-up after JIT compilation.

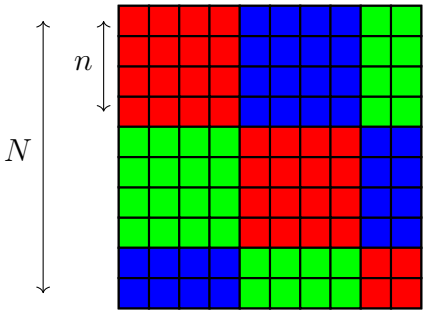
There are a number of possible loop optimisations which could be made, including identifying a loop inside a kernel, and at run-time having the loop bound be hard-coded to remove the need to evaluate the expression of every iteration.

Another possible optimisation would be Loop Fusion 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.

There is also current research into automatically 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 $n \times n$ (see Figure 29), as long as this does not violate any data dependencies in the order of operations. Doing so prevents values from being evicted from L1 cache prior to being needed again.

Figure 29: 2D Loop Tiling



Currently this has only been applied to OPS [30], the precursor to OP2 [28], which supports structured mesh solvers only, but there does exist a 2019 paper [35] on automated loop tiling for unstructured meshes, and the issues posed by the need for indirect array accesses. A library is provided which demonstrates the technique [8], including a demo using the same *airfoil* application used for this report.

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, so it would not have much impact on the results gathered.

Using CUDA's native JIT compilation pipeline would provide the added benefit that an application developer using OP2 would not have to write the Makefile themselves, as currently as 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

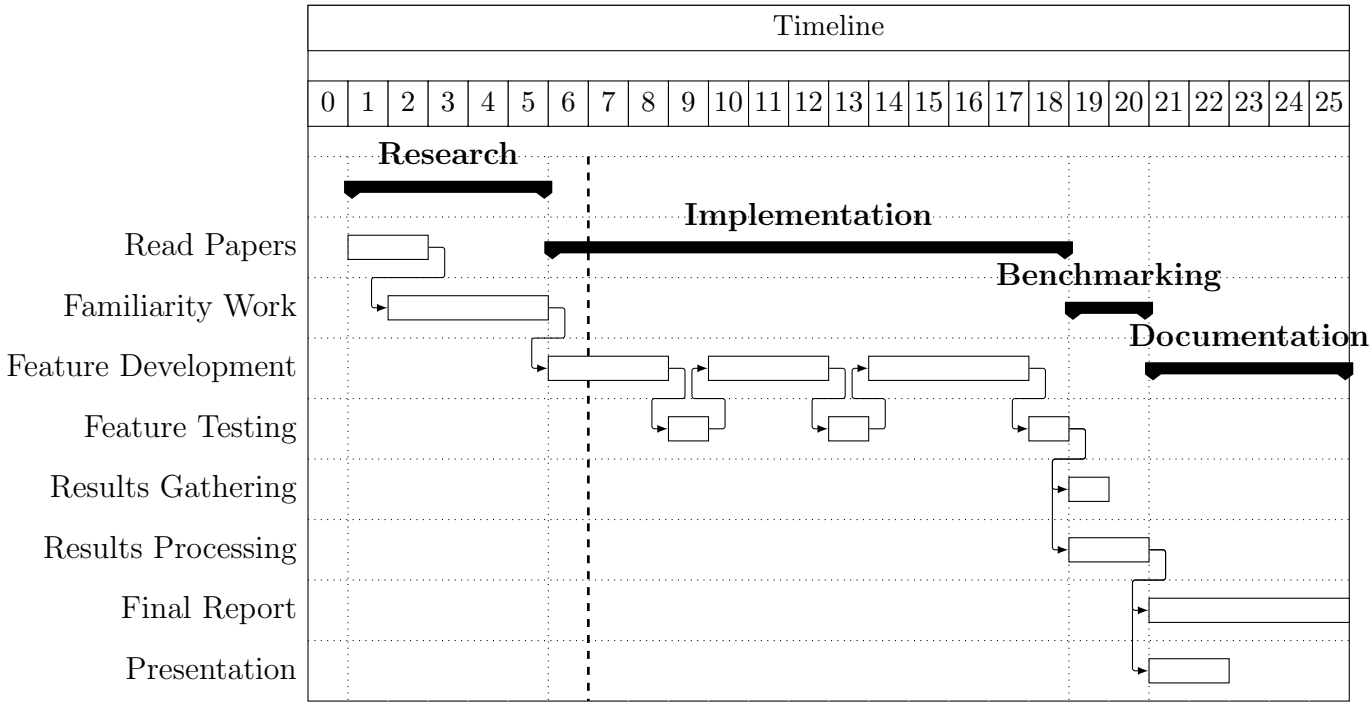
Finally, there are other hardware targets supported by OP2 which may be able to benefit from Just-In-Time compilation, and since the purpose of OP2 is to provide performance on multiple hardware platforms from a single application code any new optimisation which is found to improve performance should also be ported to other platforms, where it might be able to provide benefit. Users who do not primarily execute applications on NVidia GPU hardware should still have the option to utilise 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. A breakdown of the time investment distribution can be found in Appendix C (p95).

The Gantt chart in figure 30 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 30: 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; as well as working hands-on with both CUDA and OP2 to try to build familiarity before the actual implementation began. On

reflection, my research was mostly focused on the existing OP2 work, and many of my sources were from the same authors.

Once I had already decided on my approach and begun the implementation I discovered some similar work which might have influenced the direction of the project if I had been aware of it earlier on. For example, the *easy::JIT* approach of performing a code generation stage at runtime as well as compiling is an interesting direction.

The time I had allotted to research did need to be extended from the plan at the outset of the project, as having originally given myself just three weeks I was not confident enough with the existing OP2 code base to begin to contribute. An additional two weeks to make the five shown in Figure 30 were sufficient for me to feel able to begin the implementation.

Considering that I had never written code for a graphics card before starting this project, I am happy with the confidence in using the CUDA programming model I have built.

2. Implementation

The Implementation progressed largely as expected, with advancements made at a good pace for the time allocated. I did find that while Figure 30 lists a total of 3 full weeks for testing, partial solutions were difficult to test fully, as there is no executable generated with which 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 compiling the manually

written code to figure out how it could work. Once it was compiling successfully, the code generator could be modified to produce equivalent code, but with application specific names replaced with variables so that it would work for any application.

I 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 the previous Section 6.1. Since the core functionality of the project was only completed with two weeks of planned implementation time left (16 weeks in), it was decided that this was unlikely to be completed in the time frame, and that 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 for the project as a whole that the decision to move on to benchmarking in week 17 instead of pursuing further functionality was taken, as the original plan allotted only one week for gathering results, and a second week for analysing them. In reality, the extra two weeks that had been allocated to Implementation were also required, as well as an additional week that was supposed to be used for making the project presentation. Benchmarking was completed 21 weeks into the project, although further data was gathered after the presentation for this report.

The delay was mostly due to the desire to use a 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 could allow me access to a Kepler generation GPU, and getting familiar with using the system once accepted, took longer than expected.

With the benefit of hindsight it is clear that this process should have begun at

the start of the project, as it was always going to be necessary to have access to a HPC cluster, and be familiar with using it by the time the implementation was ready for benchmarking.

Eventually the Cambridge Service for Data-Driven Discovery (CSD3) kindly approved for me to use their *Wilkes2* GPU cluster, with workloads I submitted being placed in a 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 had failed with some error. As with many supercomputer clusters, *Wilkes2* requires jobs to be submitted using SLURM [45]. I was already starting to become familiar with SLURM from using it as part of a High Performance Computing module, and my knowledge only improved for needing to make use of 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 [25], and I believe it went well, although the demonstration was perhaps not as thorough as it should have been and perhaps did not represent my work to its fullest. The benchmarking results I collected for the presentation were only for 10,000 time steps, while the results in this report ran for 1,000,000 - which gave a much better indication of the outcome of applying the optimisation. The total execution time of the application increased from 30 seconds to 40 minutes, which allows a small change to amplify into a more noticeable difference.

This report was produced as the final submission and utilises LaTeX and BibTeX. It was completed well within the allotted time, allowing for sufficient re-drafting and feedback. The two week extension provided to account for the current pandemic was helpful to make the final stages of the report less stressful to write.

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. Since there was no collaboration, the workflow is simple and there is only a single branch, with commits whenever a feature or fix is completed.

For development, the use of GNU make for AOT compilation is sufficient,and very convenient 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, as they would need to recreate the contents of my Makefile themselves. It also relies on the OP2 library files, the generated code and the Makefile all being accessible by the binary at run-time, which might not always be the case. This is definitely something that could be improved upon by using NVidia's native CUDA JIT compilation.

Code was mostly written using Atom [3] as a text editor, or Vim [47] when working remotely on the HPC cluster. Atom's GUI is useful when working on a code-base with many files, especially when the output is also a set of files. I used Vim when only a terminal is available, because it is the editor I am most familiar with, and because it is usually already available on Linux systems.

Lastly, Google Slides was selected for the presentation because of familiarity, and to ensure changes are automatically saved to a remote in case of loss of data. I prefer LaTeX for producing reports as it allows greater control of the structure of the document than other editors, and provides access to TikZ [40]: a powerful package used for producing diagrams.

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, a fully functioning implementation of the technique has been designed and completed, which can be applied to OP2 applications that utilise the C API.

The implementation successfully augments the existing OP2 code generation with the ability to execute JIT compiled code, and applies an optimisation that is made based on the inputs of the program: defining the constant values from the input as pre-processor literals. This could only be done at run-time, and would not be possible by the existing OP2 generated code.

The results from benchmarking showed that there is only a small speed-up to the runtime, but it would appear that the speed-up is linear and the cost is constant with respect to problem size. It is important to draw the distinction that it was the run-time optimisation of defining of constants which was only providing limited speedup; and that these results definitely prove that there is benefit to be gained from applying JIT compilation in this context.

It is likely that a further optimisation such as loop blocking will be implemented on top of this implementation in the future, to improve on the per-iteration speed-up. Adding additional run-time optimisations to OP2 will be easier as a result of the work completed for this project, as they will be extensions to this implementation .

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] Krste Asanovic et al. "The Landscape of Parallel Computing Research: A View from Berkeley". In: *EECS Department, University of California, Berkeley* EECS-2006-183 (December 2006).

[3] *Atom: A hackable text editor for the 21st Century.*

URL: `https://atom.io/` (visited on 05/02/2020).

[4] *Cambridge Service for Data-Driven Discovery (CSD3).*

URL: `https://www.hpc.cam.ac.uk/high-performance-computing` (visited on 01/05/2020).

[5] The University of Cambridge. *Wilkes2 Supercomputer.*

URL: `https://www.hpc.cam.ac.uk/systems/wilkes-2` (visited on 01/05/2020).

[6] *Clang: a C language family frontend for LLVM.*

URL: `https://clang.llvm.org/` (visited on 04/15/2020).

[7] *Compiler Optimisations: Constant Folding.*

URL: `http://compileroptimizations.com/category/constant_folding.htm`.

[8] coneoproject. *SLOPE.*

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

[9] Intel Corporation. *Intel® Xeon® Processor E5-2650 v4.*

URL: `https://ark.intel.com/content/www/us/en/ark/products/91767/`

intel-xeon-processor-e5-2650-v4-30m-cache-2-20-ghz.html (visited on 05/01/2020).

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

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

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

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

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

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

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

[17] *dlopen - Linux man page*.
URL: https://linux.die.net/man/3/dlopen (visited on 05/02/2020).

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

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

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

[21] 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).

[22] 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).

[23] 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).

[24] 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.

[25] *Google Slides*. URL: `https://www.google.co.uk/slides/about/` (visited on 05/02/2020).

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

[27] 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).

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

[29] 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).

[30] 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).

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

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

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

[34] 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.

[35] Fabio Luporini et al. “Automated Tiling of Unstructured Mesh Computations with Application to Seismological Modeling”. In: *ACM Transactions on Mathematical Software* 45 (August 2017). DOI: 10.1145/3302256.

[36] 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).

[37] 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).

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

[39] Oracle. *About Java*.
URL: https://go.java/?intcmp=gojava-banner-java-com (visited on 08/05/2020).

[40] Overleaf. *TikZ Package*.
URL: https://www.overleaf.com/learn/latex/TikZ_package (visited on 05/02/2020).

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

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

[43] 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.

[44] 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.

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

[46] *The OpenMP API specification for parallel programming.*

URL: `https://www.openmp.org/` (visited on 04/27/2020).

[47] *Vim: the editor.*

URL: `https://www.vim.org/about.php` (visited on 05/02/2020).

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 05 09 02
  02 08 09 07 03 06 01 02 09 03 01 09 04 07 08 04 05 00 03 06 01 00 06 03 02 00 06 01 05 05 04 07


  05 14 16 12 06 11 07 04 18 04 03 16 04 16 11 10 05 06 05 12 02 08 13 12 04 00 08 04 12 10 13 09
```

B Getting Started with OP2

The following is a guide for getting started with the *airfoil* application, using the JIT compilation feature documented by this report.

B.1 Source Code

Firstly, the source code needs to be retrieved from GitHub, you can do using the following commands:

```
> git clone https://github.com/OP-DSL/OP2-Common.git
> cd OP2-Common/
> git checkout feature/jit
```

This puts you in the feature branch for the project. `git log` should give the latest commit as:

```
commit 92865104e33e4448a180217a994b6cb1178bfab8
Author: NDunne <24ndunne24@gmail.com>
Date:   Fri May 8 12:51:08 2020 +0100

    Updated READMEs
```

If it does not, then more work may have been added to the branch, and the commit ID will be need to be checked out instead:

```
> git checkout 92865104e33e4448a180217a994b6cb1178bfab8
```

This will leave you in a detached head state, but this is ok since we are just using the files, not modifying them for a future commit.

After obtaining the source code, check the contents of the current folder (`OP2-Common/`):

```
> ls
apps/  AUTHORS  cmake/  doc/  LICENSE  op2/  README  scripts/  translator/
```

To double check that you do indeed have the JIT compilation feature, run the following command and confirm the result:

```
> ls translator/c/python/jit/
__init__.py  op2_gen_cuda_jit.py  op2_gen_seq_jit.py
```

B.2 System Setup

As well as the OP2 source, we also require a number of dependency libraries, and need to set some environmental variables. I recommend adding these to a setup script, or to `~/.bashrc` to save time in the future.

B.2.1 3rd Party Libraries

The OP2 framework relies on a couple of additional libraries:

- C/C++ Compiler e.g. Intel *icc*
- HDF5
- ParMetis
- PT-SCOTCH
- GNU Make
- MPI Implementation e.g. Intel MPI
- CUDA Toolkit

If you are using a HPC cluster check if these are available using the `module` command. If they are not, you will need to download and build each one.

B.2.2 Environmental Variables

Various Makefiles used by OP2 expect the below Environmental Variables to be set.

An example `~/.bashrc` script is show below, which gives example values for each.

```
#OP2 env variables

unset LD_LIBRARY_PATH

export OP2_COMPILER=intel
export OP2_INSTALL_PATH=/home/ndunne/Documents/OP2-Common/op2/

alias op2="python2 $OP2_INSTALL_PATH/../translator/c/python/op2.py"

export PARMETIS_INSTALL_PATH=/opt/parmetis-intel/
export PTSCOTCH_INSTALL_PATH=/opt/scotch-intel/
export MPI_INSTALL_PATH=/opt/mpich/
export HDF5_INSTALL_PATH=/opt/phdf5-intel/
export CUDA_INSTALL_PATH=/opt/cuda/

export LD_LIBRARY_PATH=/opt/parmetis-intel/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/opt/scotch-intel/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/opt/phdf5-intel/lib:$LD_LIBRARY_PATH

#CUDA
export PATH=/opt/cuda/bin:$PATH
export LD_LIBRARY_PATH=/opt/cuda/lib64:$LD_LIBRARY_PATH
export CUDA_INSTALL_PATH=/opt/cuda

#INTEL MPI
export MPI_INSTALL_PATH=/opt/intel/impi_latest/intel64/
source /opt/intel/bin/compilervars.sh intel64
source $MPI_INSTALL_PATH/bin/mpivars.sh intel64

export INTEL_PATH=/opt/intel
export MPICH_CXX=$INTEL_PATH/bin/icpc
export MPICH_CC=$INTEL_PATH/bin/icc
```

Once these have been set, you can compile the OP2 libraries and translate an application.

B.2.3 OP2 Libraries

With environmental variables set the next stage is to compile the OP2 library files. There is a different target for each hardware platform, but for this implementation the required targets are `core`, `cuda`, and `hdf5`. Other back-ends are required for the tool to convert the input mesh to HDF5, so the command will just be `make`, to build them all.

```
> cd op2/c/
> make
```

B.2.4 Airfoil

The *airfoil* source code can be found in the OP2 repository, by returning to the top level folder, and traversing to `apps/c/airfoil/airfoil_JIT/dp/`.

```
> cd ../../
> cd apps/c/airfoil/airfoil_JIT/dp/
> ls
adt_calc.h  airfoil.cpp  bres_calc.h  convert_mesh.cpp  convert_mesh_mpi.cpp
Makefile  res_calc.h  save_soln.h  update.h
```

The input data file will need to be downloaded from the OP2 Website into this directory, and as it is not already in the HDF5 data format, the *convert_mesh* tool will need to be built, and the file converted.

```
> wget https://op-dsl.github.io/docs/OP2/new_grid.dat
> make convert_mesh_seq
> ./convert_mesh_seq new_grid.dat
> mv new_grid_out.h5 new_grid.h5))
```

Alternatively the MPI version can be used, by building the target `convert_mesh_mpi`.

The result will be a file `new_grid.h5`. You can view the contents using `h5dump`.

Code Generation

It is time to perform the code generation step. As in seen in Section 5.3, the command is as follows, as long as the `$OP2_INSTALL_PATH` environmental variable is correctly configured.

```
> python2 $OP2_INSTALL_PATH/../translator/c/python/op2.py airfoil.cpp JIT
> ls
adt_calc.h      bres_calc.h     convert_mesh_seq     new_grid.dat
save_soln.h     airfoil.cpp     convert_mesh.cpp     cuda/
new_grid.h5     seq/            airfoil_op.cpp       convert_mesh_mpi.cpp
Makefile        res_calc.h      update.h
```

The generated code is now present, and can be seen in the folder named `cuda/`. It is ready to be compiled into an executable.

Compilation

Using the Makefile to compile, both a JIT enabled and disabled binary can be produced easily. For JIT compilation enabled:

```
> make airfoil_cuda
> ./airfoil_cuda_jit
```

And for disabled:

```
> make airfoil_cuda JIT=FALSE
> ./airfoil_cuda
```

And the binaries should run as expected, and pass the test after 1000 iterations.

B.3 Other Applications

The translator script should work just as well with other applications written for OP2, however they will need to reproduce the Makefile in order to work correctly. It is required that the Makefile is accessible by the running application for JIT compilation to work correctly, and must contain a target similar to the following:

```
airfoil_cuda_rec: jit_const.h
  nvcc $(VAR) $(INC) $(NVCCFLAGS) $(NVCC_COPTS) $(OP2_INC) -Icuda -I. -c \
  ./cuda/adt_calc_kernel_rec.cu -o ./cuda/adt_calc_kernel_rec.o
  nvcc $(VAR) $(INC) $(NVCCFLAGS) $(NVCC_COPTS) $(OP2_INC) -Icuda -I. -c \
  ./cuda/bres_calc_kernel_rec.cu -o ./cuda/bres_calc_kernel_rec.o
  nvcc $(VAR) $(INC) $(NVCCFLAGS) $(NVCC_COPTS) $(OP2_INC) -Icuda -I. -c \
  ./cuda/res_calc_kernel_rec.cu -o ./cuda/res_calc_kernel_rec.o
  nvcc $(VAR) $(INC) $(NVCCFLAGS) $(NVCC_COPTS) $(OP2_INC) -Icuda -I. -c \
  ./cuda/save_soln_kernel_rec.cu -o ./cuda/save_soln_kernel_rec.o
  nvcc $(VAR) $(INC) $(NVCCFLAGS) $(NVCC_COPTS) $(OP2_INC) -Icuda -I. -c \
  ./cuda/update_kernel_rec.cu -o ./cuda/update_kernel_rec.o   \
  nvcc cuda/adt_calc_kernel_rec.o cuda/bres_calc_kernel_rec.o \
  cuda/res_calc_kernel_rec.o cuda/save_soln_kernel_rec.o      \
  ccuda/update_kernel_rec.o -shared -o ./cuda/airfoil_kernel_rec.so
```

The target compiles each of the JIT kernel files into an object file, then combines all the objects into a single Shared Object called `cuda/airfoil_kernel_rec.so`. The name of the resultant DLL must follow this name pattern, otherwise it will not be found by the JIT enabled binary.

C Time Investment

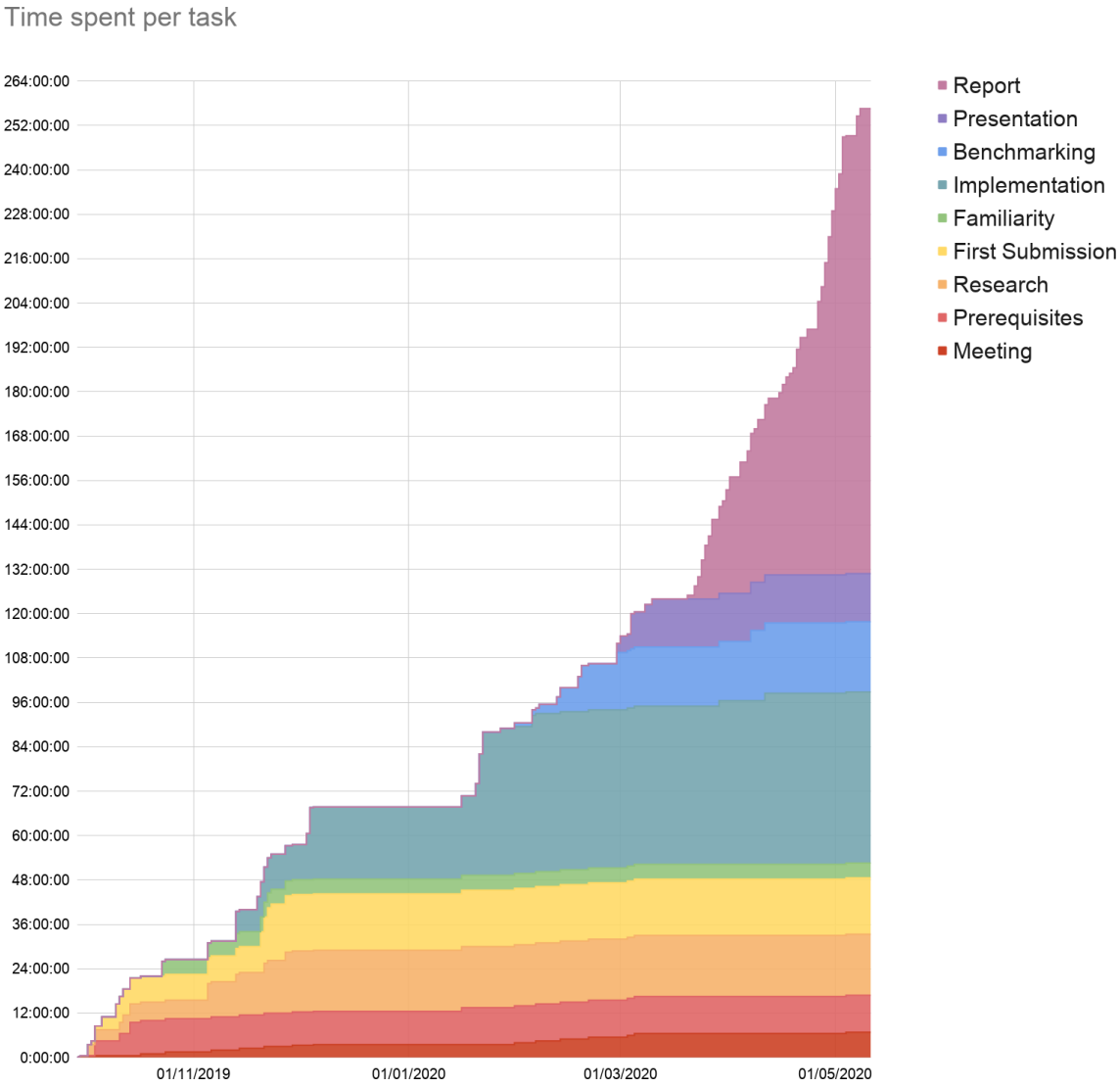


Figure 31: Time Investment broken down by task.

While working on this project I attempted to track all the time I put into it, in an attempt to hold myself accountable and ensure work was done at a reasonable rate. The result is the graph above, plotted stacked to track total time investment also.