Just In Time Compilation for a High-Level DSL

Nathan Dunne

1604486

3rd Year Dissertation

Supervised by Gihan Mudalige

Department of Computer Science

University of Warwick

2019–20

Abstract

TODO

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

Key Words

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

Contents

List of Figures

1 Introduction

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

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

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

1.1 Background Work

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

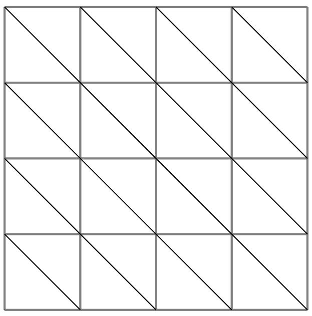


Figure 1: Tri-Structured Mesh

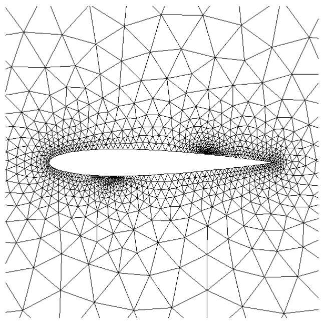


Figure 2: Airfoil Tri-Unstructed Mesh

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

1.2 Motivations

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

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

2 Research

2.1 OP2

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

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

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

2.2 CUDA

2.3 Related Work

2.3.1 JIT

2.3.2 Similar Libraries

3 Specification

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

The Python script will perform source-to-source translation. As input it will take the application files, which specify the structure of the program: declaring variables, and indicating where the loops parallelised by OP2 should be executed; and a kernel descriptor for each loop, which will describe the operation to be performed on each element of the set passed to the loop as a parameter. OP2 makes an important restriction that the order in which elements are processed must not affect the final result, to within the limits of finite precision floating-point arithmetic[3, p3]. This constraint allows the code generator freedom to not consider the ordering of iterations, and select an ordering based on performance.

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

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

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

3.1 Runtime Assertions

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

3.2 System Model

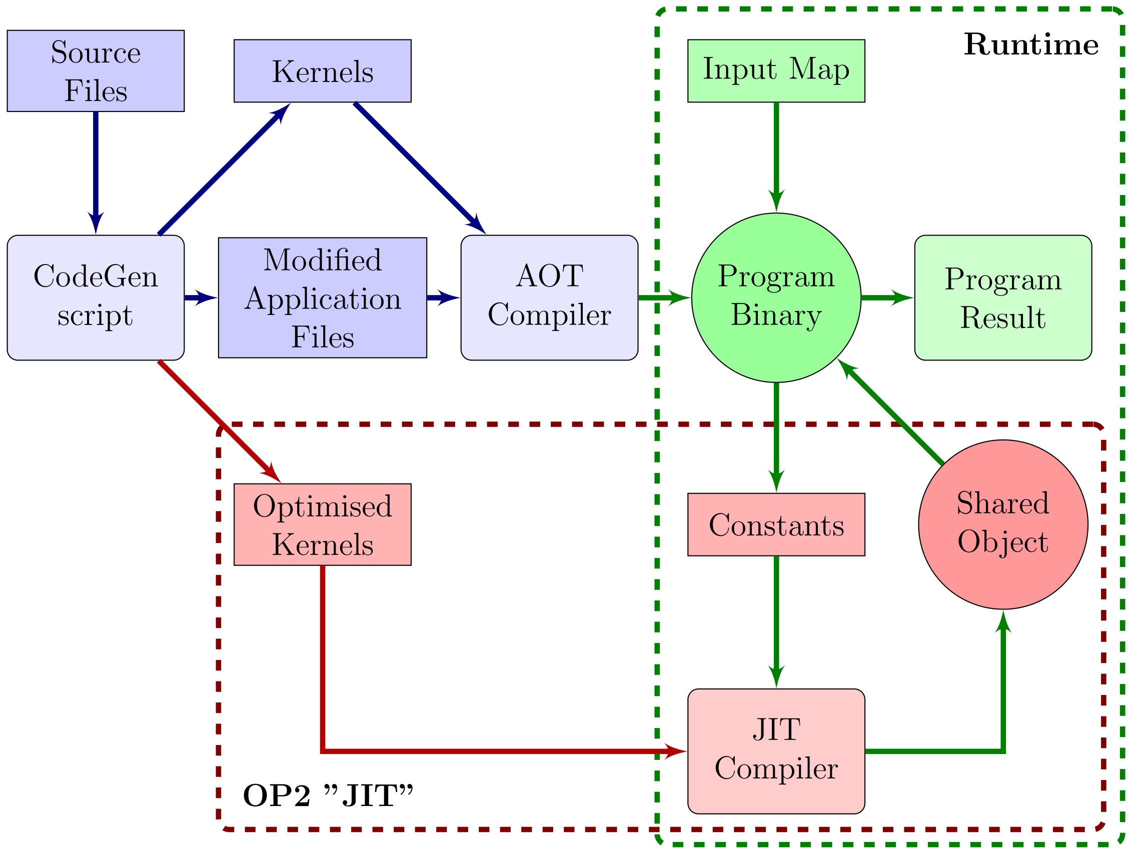


Figure 3: OP2 System Diagram with JIT Addition

Figure 3 describes the new workflow of the OP2 library, with the addition of Just-

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

The OP2 API function:

void op_decl_const(**int** dim, **char** ∗type, T ∗dat, **char** ∗name) [2, p9]

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

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

4 Implementation

The OP2 library is hosted open source on GitHub[1]. Instructions for obtaining the implementation completed for this report, and getting started with OP2 can be found in Appendix A.

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

The `lazy-execution` branch contained the beginnings of a system to execute parallel loops when values are required rather than when called. This is done through an internal library function:

void op_enqueue_kernel(op_kernel_descriptor *desc)

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

Which currently simply executes the function straight away. This process for calling parallel loops is used similarly throughout work done to enable Just In Time Compilation for CUDA, so that future efforts towards lazy execution can continue in the future on top of the JIT implementation.

4.1 Code Generation

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

def op2_gen_cuda_jit(master, date, consts, kernels)

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

Which is called from `op2.py` in the parent directory - the same as the other code generation scripts. It's parameters are:

master:	The name of the Application's master file
date:	The exact time of code generation
consts:	list of constants, with their type, dimension and name
kernels:	list of kernel descriptors, where each kernel is a map containing many fields describing the kernel, which may alter the way the code for that loop is generated.

The code generator first performs a quick check across all kernels to see if any use the Struct of Arrays feature [2, p9], or if all are using the default data layout. Then iterates over each kernel, and generates both the Ahead Of Time (AOT) kernel file, and the Just In Time (JIT) kernel file simultaneously. A folder `cuda` is created if it doesn't exist, and two files generated in it for each kernel:

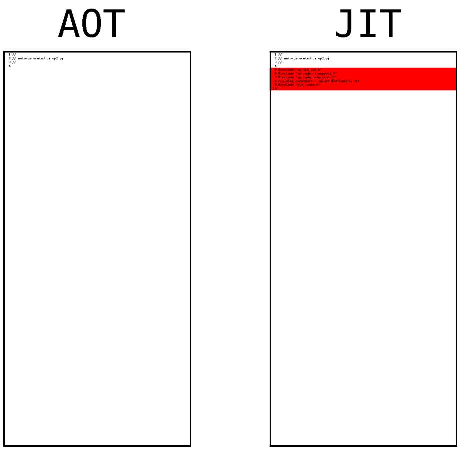
- `AOT: cuda/[name]_kernel.cu`
- `JIT: cuda/[name]_kernel_rec.cu`

Some sections of both files are the same. The examples in Figure 4 show the progression of each file for an example kernel.

The first thing generated is simply the include directives required for the JIT compiled kernel. These are needed since they will be compiled seperately:

```
#include "op_lib_cpp.h"
#include "op_cuda_rt_support.h"
#include "op_cuda_reduction.h"
//global_constants
#include "jit_const.h"
```

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



```
//
// auto-generated by op2.py
//

```

```
//
// auto-generated by op2.py
//

#include "op_lib_cpp.h"
#include "op_cuda_rt_support.h"
#include "op_cuda_reduction.h"
//global_constants - values #defined by JIT
#include "jit_const.h"

```

Figure 4: JIT includes

5 Testing

6 Evaluation

7 Future Work

8 Conclusion

References

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

[2] M.B. Giles G.R. Mudalige I. Reguly. *OP2 C++ User Manual*. `https://op-dsl.github.io/docs/OP2/OP2_Users_Guide.pdf`. 2013.

[3] 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.

Appendices

A Getting Started with OP2

Acknowledgements

TODO