EXTENDING JUST-IN-TIME COMPILATION FOR OP2

PROGRESS REPORT

Nathan Dunne - u1604486

Introduction

The aim of this project is to contribute to the OP2 open-source project[2], an Embedded Domain Specific Langauge for developing unstructured mesh based applications. It provides a programmer access to greater performace using an abstraction from hardware specific optimisations, OP2 also provides the further benefit of allowing portability between different platforms, allowing all programs written using the API to utilise the latest hardware generation by simply updating the OP2 code generation.

On top of the existing work, there is space for greater performance to be gained through the use of Just-In-Time (JIT) compilation: re-compiling the source code at run-time once the inputs are known, to allow expensive repeated operations to be replaced with static constants where possible, expanding the ability of the compiler to optimise through constant propogation and other methods. This can provide significant benefit, especially over a large number of iterations as a small improvement per-iteration can compound to better offset the one-time cost of the re-compilation.

The inputs to the program will be the description of the unstructured mesh as an input file, which is read at runtime. Since this information is not known to the compiler ahead of time, it is only possible to optimise based on this information ”Just In Time”, or immediately before execution.

The sequential JIT code generator has already been completed, so the primary goal is to target the NVIDIA GPU architecture using CUDA, attempting to reach a performance improvement over Ahead-of-Time compiled CUDA generation. Since the OP2 API itself will not be modified by the project, existing applications can be used to benchmark performance. Primarily this will be Airfoil [9], an industrially representative computational fluid dynamics (CFD) program, and later VOLNA[8], a non-linear shallow water equation solver. Both use the C OP2 API.

Motivations

The motivations behind this project are a desire to use and expand on knowledge of optimisations and high-performance computing gained in the second year *Advanced Computer Architecture* module, as well as an interest in compilers and code generation. It will also be personally beneficial to gain experience contributing to a real open-source project, with an existing code base to deal with. It is gratifying to know that if the project is completed to a high standard, it could be merged into the master branch and used beyond the submission for a module.

The decision to target GPUs is as a result of the increased need for parallelism in high performance computing systems and applications. GPUs are in nature aimed at graphics processing, which requires many operations to be performed in parallel, but similarly many HPC programs (including unstructed mesh apps like CFD programs) can benefit from this architecture now that they are capable of double precision.

Research

In order to begin contributing to the OP2 source, I needed to build an understanding of the existing work. There are 2 main sources for this information:

- Papers publish on the OP2 website
- Analysis and Benchmarking of existing source code

While academic papers are useful to understand the context of the project and the motivations behind it [7] [6] [10]; the existing code gives a more concrete understanding of the implementation, and the form my contributions will take. Understanding the basic structure of an OP2 program, where data is divided into sets and mappings between them - and loops can be defined as a seperate ”kernel” of operations, is critical to making a useful contribution.

Technical Content

As well as the research above, the following work has been completed so far.

Prerequisites and Tools

OP2 utilises two well established parallel mesh partitioning libraries: ParMetis[3] and PT-Scotch[4]. As I intend to work mainly on my personal laptop, a version of each had to be retrieved and built with the correct build arguments. The work with CUDA also required the CUDA Software Development Kit[1], and drivers for the Nvidia MX250 card I have available. Finally, the airfoil example application makes use of HDF5 file I/O operations, so a build of that library was required as well.

The project is hosted on GitHub, with a branch created to manage my contributions. This will allow me some saftey in the case of

CUDA familiarity

Since the CUDA c++ API is not something I have used before, I experimented with writing small programs to run on a gpu, as well as referencing the *Nvidia CUDA C Programming Guide* to build my Understanding. Knowledge of how CUDA programs can solve common problems in parallelism, such as local and global shared memory, and the use of *syncthreads()* will undoubtably be necessary to complete this project.

Hardware Resource

The graphics card in my personal laptop is sufficent to execute the generated CUDA code, and ensure correctness, however the system will be too noisy to gather representative benchmarks. To this end, I applied for and have been granted access to the Orac Hight Performance Computing node in the department, which will be used only for benchmarking.

Existing code familiarity

The existing JIT work was done in the branch *lazy-execution* of the Git repository. In order to extend this work, I've rebased the branch onto the master branch in order to benefit from the many changes commited to master since they diverged.

Once this was done, I could begin to understand what the existing code does and how my code fits in. I was already aware that the bulk of my work will be done in the ”translator” folder, with C as the target and python as the script language. Starting here with the main *op2.py* script makes sense. This script parses the input file for API calls such as *op_decl_set()*, *op_decl_const*, and *op_par_loop()*; for declaring sets, constants and loops respectively. These are sections that will need to be replaced for a normal compiler to accept them.

Each one is checked to be well formed, and the information described in their parameters recorded. This is especially important for loops, where the number of arguments and the type of each argument is defined, as well as whether it is optional, and stored in the list of ”kernels”. In order to be valid, each kernel must also have a definition of the operations to perform on each iteration, which for the example application *airfoil* is a seperate header file of the same name, but this is not required.

The original input file is then written out, with declarations for functions representin each loop with the right number of parameters, and in the place of the loops: calls to these function. The definitions will be generated by the code generation scripts.

op2.py then calls each codegen script for all the supported backends, to generate a folder for each backend containing the generated code. For JIT, initially only op2_gen_seq_jit existed, so my contribution will be to populate this list of code generators similar to the Ahead-of-Time list. The reason for all this analysis of the op2.py script, is to understand the state of the file when the code I write is called, and the parameters that it will recieve:

1. masterFile: the name of the first file passed to op2.py
2. date: the date today
3. consts: master list of declared constants in source file
4. kernels: master list of loop kernels declared in source file

The code generator can then loop through the kernels, and generate a function to call for each one based on the target hardware.

Implementation

At time of writing the implementation of CUDA JIT code generation has just started. The op2.py script has been modified to call a CUDA_jit code generation file, and added to the makefile of airfoil to include an airfoil_CUDA_jit entry, however the codegeneration itself is still sequential. To complete the CUDA JIT implentation I will use the AoT CUDA implementation as a guide to speed up the process. The work is available in my *feature/jit* branch of the remote repository.

Reflection and Project Management

The timetable from the project specification (Figure 1) shows the expected level of progress at this stage. In retrospect, It is clear that the original timetable was ambitious, as the project has not run quite to time. While I am still confident that the CUDA JIT code generation can be done by the end of the end of the autumn term (as originally agreed with my supervisor), I have reworked the timetable with a greater allocation of time given to research, and three iterations of longer implementation blocks with one week testing, rather than five rounds of 3 week blocks. Now having a greater understanding of the project, I think it unreasonable to suggest there would a feature worth spending a week testing and benchmarking completed every 2 weeks of development. See Figure 2. As well as aligning with the end of the Autumn term, this also allows for two more weeks development of time overall.

I would attribute the need for an extension to research time to taking an ad hoc approach to working on the project, which has resulted in not putting as much time in as I had planned, instead focussing on closer deadlines. As such I have decided that for next term I will set a weekly timetable of allocated time to work on the project.

The timetable aside, other elements of the project haven't provided any significant unexpected issues.

Ethics and Risks

As mentioned in the Project Specification, there are no obvious social or ethical issues with the project, as it will not require the collection or storage of any personal data. It is important to consider the lisence under which the open source project is held, which permits redistribution of the source and binary, as long as it contains the copyright disclaimer.

I am also now aware of the Acceptable Use Policy[5] for the computing facilities provided bt the SCRTP, so I will also ensure to abide by this when benchmarking using Orac.

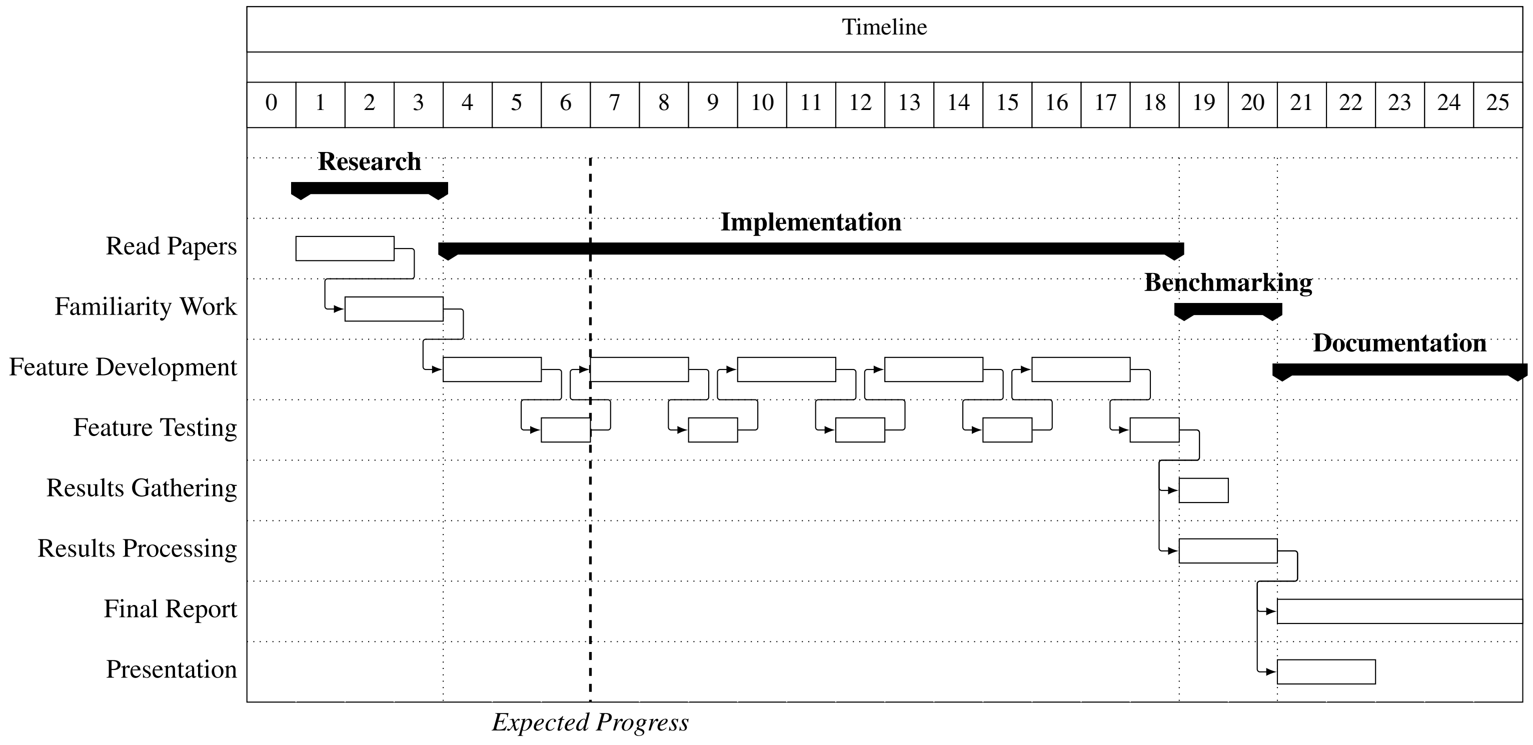

Figure 1: Original Timetable

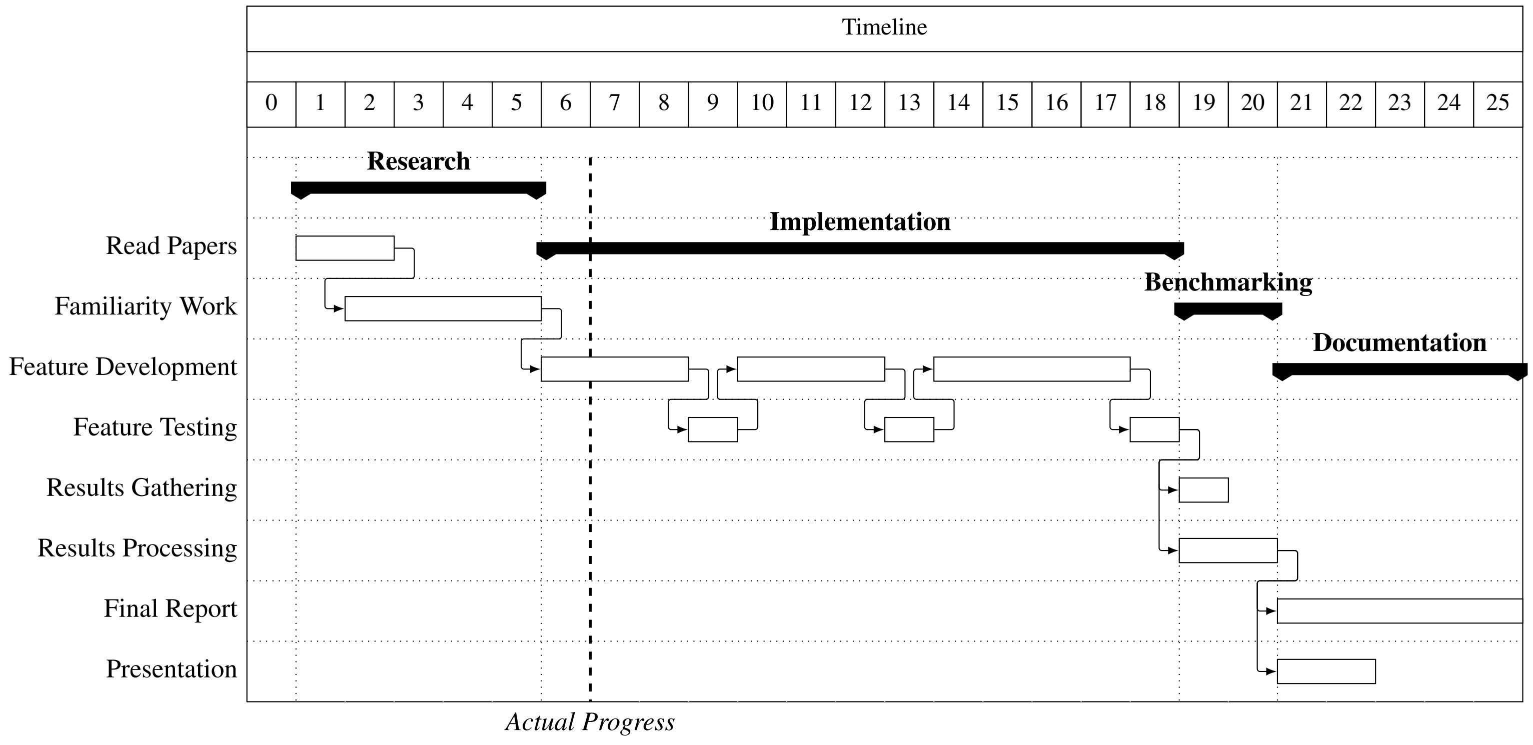

Figure 2: Modified Timetable

References

[1] Cuda sdk. https://developer.nvidia.com/cuda-toolkit.

[2] Op2-common. https://github.com/OP-DSL/OP2-Common.

[3] Parmetis. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[4] Pt-scotch. https://www.labri.fr/perso/pelegrin/scotch/.

[5] Scientific computing rtp acceptable use policy. https://warwick.ac.uk/research/rtp/sc/policies/aup.

[6] I.Z. Reguly G.R. Mudalige and M.B. Giles. Auto-vectorizing a large-scale production unstructured-mesh cfd application, 20. https://www.oerc.ox.ac.uk/sites/default/files/uploads/profile-pages/Gihan/GRM-WPMVP.pdf.

[7] 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, 20. https://www.oerc.ox.ac.uk/sites/default/files/uploads/profile-pages/Gihan/InPar20.pdf.

[8] et al. Istvan Z. Reguly, Daniel Giles. The volna-op2 tsunami code, 2018. https://www.geosci-model-dev.net/11/4621/2018/gmd-11-4621-2018.pdf.

[9] Istvan Reguly Mike Giles, Gihan Mudalige. Op2 airfoil example, 2012. https://op-dsl.github.io/docs/OP2/airfoil-doc.pdf.

[10] I.Z. Reguly et al. Acceleration of a full-scale industrial cfd 30 application with op2, 20. https://www.oerc.ox.ac.uk/sites/default/files/uploads/profile-pages/Gihan/OP2-Hydra.pdf.