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, an Embedded Domain Specific Langauge for describing unstructured mesh based applications. It provides an abstraction from the hardware to allow hardware specific performance benefits, such as SIMD vectorisation and GPU high levels of parallelism, to be gained without the programmer needing to understand the latest implementations, as well as allowing portability between different systems. The API has the added benefit of only having to update OP2 to include support for the newest hardware optimisations, and all applications written using it will be able to benefit.

On top of the existing work, there is space for greater performance to be gained through the use of Just-In-Time compilation: re-compiling the source code at run-time once the parameters are known, to allow expensive operations to be replaced with static constants where possible. This can provide significant benefit, especially over a large number of iterations as a small improvement can compound to better offset the one-time cost of the re-compilation. The sequential implementation has already been done, so the primary goal is to target the NVida GPU architecture using cuda, attempting to reach a performance improvement over Ahead-of-Time compiled cuda generation.

Since the OP2 API is in use, and the API itself will not be modified by the project, existing applications can be used to benchmark performance. Primarily this will be airfoil, an industrially representative computational fluid dynamics (CFD) program.

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 bae 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.

Research

In order to begin contributing to the OP2 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 of existing source code

While academic papers are useful to understand the context of the project and the motivations behind it; the existing code gives a more concrete understanding of the implementation, and the form my contributions will take. Under-

standing 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.

There is a significant amount of literature already on performance engineering for GPUs, so it will mostly be a case of implementing the theory rather than investigating the possibility of performance benefit from further parallelism. The case for Just-In-Time compilation is made by its applications for performance from Java to LLVM.

Technical Content

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

Prerequisites

OP2 utilises two well established parallel mesh partitioning libraries: ParMetis and PT-Scotch. 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 NVidia toolkit 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.

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.

Timetable

Plan

Reflection

Ethics

Project Management

*

References

*

Further Reading