OP2-clang

December 15, 2017

Contents

1 Introduction

OP2 is a high level framework with associate libraries and preprocessors to generate parallel executables for unstructured mesh applications. This document describes the usage and implementation of the clang based C++ preprocessor for OP2.

2 Overview

OP2-clang is a clang based preprocessor using clang's LibTooling to generate parallel executables. Currently OP2-clang enables users to write single program which can be built into three executables:

- Single-threaded on a CPU
- Multi-threaded using OpenMP for multicore CPU systems
- AVX vectorized for x86 CPU systems

OP2-clang is a clang based "source-to-source translator" tool. It's purpose is to generate parallelised and optimised source code for unstructured-mesh applications using OP2's abstractions. OP2-clang provides a modular abstraction to perform the translation and gives a convenient interface to extend it with new optimisations and parallelisation approaches. This tool is operating on the AST of the application. It collects data about the application from based on the OP2 library calls and with simple replacement on the main sources it generates copies for parallel execution. Then based on the collected data the tool generates optimised sources for the parallel loops. The kernel generation is operating on parallelisation approach specific simple skeleton codes. The generator parse the skeletons for each parallel loop and alter the source code through replacements to get the corresponding kernel used to get parallel executables.

3 Build

You will need a checkout of the llvm, clang and clang-tools-extra source code first (see `http://clang.llvm.org/get_started.html` for instructions). Check out the OP2-Clang repository and set `OP2_INSTALL_PATH` e.g. with:

```
git clone https://github.com/bgd54/OP2-Clang.git;
export OP2_INSTALL_PATH="/path/to/op2/"
```

Then in the OP2-Clang directory:

```
mkdir build
cd build
cmake ..
make
```

4 Usage

The tool can be run on the command-line with the command:

```
./op2-clang main.cpp sub1.cpp -- clang++ --app-spec-flags
```

Assuming that the application is split over several files. This will lead to the output of the following files `main_op.cpp, sub1_op.cpp, sub2_op.cpp` and a master kernel file (`main_xxxkernels.cpp`) for each approach generated where xxx is differ between executables, and an individual kernel file for each parallel loop.

- The `main_op.cpp` file is used for all versions (sequential, OpenMP and vectorized) executables.
- The `main_xxxkernels.cpp` files are including the individual kernel files (`<loop_name>_xxxkernel.cpp`).

- The individual kernel files (`<loop_name>_xxxkernel.cpp`) containing the efficient parallelised versions of the parallel loops. These files generated from the corresponding skeletons with modifications based on the information collected from `main.cpp` (and from other files parsed during the invocation of the tool) about the loop.

The code generation can be controlled through `optarget` flag:

```
./op2-clang -optarget=vec main.cpp -- clang++
```

In this case only the version specified by the value of `optarget` will be generated.

5 Implementation

OP2-Clang performs the source-to-source translation with multiple RefactoringTools. The RefactoringTools are using clang's AST matcher API, to find interesting parts of the application in the AST and generating replacements based on the matches.
The translator consists of two main part the processing of the OP2 application and the generation of target specific codes.

5.1 Processing OP2 Applications

The first part is responsible for collecting information about the OP2 application and the generation of the modified application files, which are used to compile the target specific executables.
The information collection and the modifications are performed along the OP2 API calls in the application. With matchers on the API calls we can register declaration of sets, mappings, global variables, etc. Also the information (the operation set, the arguments, the function that applied to each set element) about all parallel loops are collected along the **op_par_loop** calls and these calls are substituted with the calls of the **op_par_loop_[loop_name]** functions that are generated for each target. The modified application files are saved to separate files with names **[filename]_op.cpp**.
After finishing the Data collection this layer invokes the target specific code generators.

5.2 Target Specific Code Generation

The second part of the translation is responsible for generating two types of target specific files:

- **[application_name]_[target]kernels.cpp**: one file for each target generated (referred to as master kernel file)
- **[loop_name]_[target]kernel.cpp**: for each parallel loop for each target generated (referred to as kernel files)

The generation of these files are also performed with RefactoringTools and replacements. The basic structure of these files are independent from the application. It only depends on the target that the file generated for, therefore we can use skeletons as an input for the RefactoringTools. The generation of code generation for one target can be break down to three layers:

- Generation of the master kernel file.
- Generation of separate kernel files.
- Generation modified versions of the user function for kernel files.

5.2.1 Master Kernel Generator

The master kernel file contains target specific defines, the declarations of global variables or the definition of **op_decl_const_char** function and includes the kernel files.
The master kernel generators is responsible for generate this file and invoke the kernel generators for each loop.

5.2.2 Kernel Generator

The kernel generators are transforming the target specific skeletons (usually there are two skeletons for a target one for generating direct kernels and one for generate indirect kernels) based on the information about the loop that currently processed in a similar way as above. Each skeleton is used as a pseudo code for the loop and the translation is basically just change the code to the data in the loop. The generator builds the AST of the skeleton and using matchers it changes the specific parts.
To use optimised versions of the user the kernel generators invoke a RefactoringTool on the user function to get a modified copy of it, which is used in the generated kernel file.

5.2.3 Modification of the user function

This layer of RefactoringTools is operating on the definition of the user function. For this a temporary header is generated with the global variables, and another file containing the original user function. Then the tool build the AST for the user function and pass the resulting string back to the kernel generator.