Exercises with SYCL/oneAPI

Escuela de Ciencias Informáticas 2022

July 25, 2022

This document contains all the lab exercises for the Heterogeneous Programming Models course at the Escuela de Ciencias Informáticas 35. The initial preliminary notes includes some comments on how to verify the setup to complete the labs. After, the document describes all the laboratories. The complexity increases with the lab numbers, so please do not attempt any exercise without having complete the previous ones.

To ensure the completion, none of the laboratories requires a physical FPGA device, which limits the testing and performance analysis. However, all of them can be run on the Intel DevCloud platform.

Happy coding! If you want to check more examples and learn from other design patterns please visit the collection of excellent oneAPI examples.

1 Preliminary notes

Please ask the instruction to grant you permissions to the class repository.

1.1 oneAPI setup

This document assumes that the oneAPI toolchain is correctly installed on your system, including the FPGA emulator.

To verify this assumption you can run any of the following two commands:

```
$ oneapi-cli # runs a command line tool with examples among others
$ dpcpp -v # runs the Intel data parallel C++ compiler
```

In case the tools are not available, please check whether `setvars.sh` has been run and consider executing the file when you log in; e.g.; in bash, add the following code to your `.bash_profile`

```
if [ -f /opt/intel/oneapi/setvars.sh ]
then
    source /opt/intel/oneapi/setvars.sh
fi
```

1.2 oneAPI FPGA simplified compilation types

oneAPI FPGA compilation flow sets the final image type, from the quick and useful to debug emulator to the real bitstream for the hardware.

```
# FPGA emulator image
$ dpcpp -fintelfpga fpga_compile.cpp -o fpga_compile.fpga_emu
```

```
# FPGA simulator image
$ dpcpp -fintelfpga fpga_compile.cpp -Xssimulation -Xsboard=intel_s10sx_pac:pac_s10

# FPGA early image (with optimization report): default board
$ dpcpp -fintelfpga -Xshardware -fsycl-link=early fpga_compile.cpp -o fpga_compile_report.a

# FPGA early image (with optimization report): explicit board
$ dpcpp -fintelfpga -Xshardware -fsycl-link=early -Xsboard=intel_s10sx_pac:pac_s10
↪  fpga_compile.cpp -o fpga_compile_report.a

# FPGA hardware image: default board
dpcpp -fintelfpga -Xshardware fpga_compile.cpp -o fpga_compile.fpga


# FPGA hardware image: explicit board
dpcpp -fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp -o
↪  fpga_compile.fpga
```

The dpcpp compiler also supports standard C++ files and can be used to compile them.

2 Laboratory 1: Initial Example, adding two vectors

The first initial example of the course is the addition of two vectors A and B of 8192 elements into the C vector. Since SYCL is based on C++ metaprogramming, you will first implement a C++ only version and then write the SYCL equivalent.

2.1 C++ version

Inside the lab1 directory of the hangouts, you have the add_array_serial_skeleton.cc file that you have to complete. For compilation, you can compile directly with cmake or from the command line. In general, the first option is preferred, but since this is a single file, you can also compile with the command line.

```
$ dpcpp -I./ -o add_array add_array_serial_skeleton.cc
```

The include flag is required to compile with the add_array.hh file header.

2.2 SYCL version

One of the most common parallel and heterogeneous programming patters in parallel for, which simultaneously runs in parallel independent iterations. To understand this pattern, we are going to translate the add array program into their SYCL equivalent using parallel_for.

For the next step, please copy your C++ serial version of the add array example into an add_array_sycl.cc file and modify the code to use SYCL. Please remember that you will need a queue, 3 buffers with their respective accessors, and a handler.

One key aspect to optimize array accesses in programming languages with pointers is aliasing. In this case, two only-read and one only-write buffers, aliasing should not be a problem, but in more complex applications composed with multiple cases, aliasing can prevent the compiler/translator to perform optimizations such as vectorization. Sycl provides a directive, restrict, that guarantee that vectors do not overlap. Please check Intel's optimization manual to add the restrict directive to your kernel.

To compile, you can use the same approaches, cmake or command line, as in previous example.

2.2.1 Self-assessment questions

- Please use the command `time` to measure the execution time of both version, which version is faster? Why? Is it running on any accelerator?

2.3 SYCL Queue device discovery

Using the `get_devices` method, print all available devices in your machine and verify where your code is running.

2.3.1 Self-assessment questions

- Based on this laboratory?, is it possible to port C++ standard code to SYCL?

3 Laboratory 2: Approximate PI with a Taylor series

The π number represents the ratio of a circle’s circumference to its diameter, and its value was already approximated before the Common Era. The ancient Greek Archimedes already devised a π approximation algorithm. In this lab, we are going to implement another approximation based on Taylor series.

With a Taylor series, π can be approximated as

$$\pi = 4\sum_{i=0}^{\infty}\frac{-1^{n}}{2n+1}$$

Figure 1: pi Taylor approximation formula

For this lab session, the goal is first to implement the computation of pi in C++ to get familiar with the problem, and then in SYCL to deepen the knowledge of the parallel_for pattern.

3.1 C++ version

Inside the lab2 directory of this repo, there is a file called `pi_taylor_serial.cc`. Open the file and complete the body of the pi_taylor function with the aforementioned formula for a given number of steps. Instead of computing from 0 to infinite steps, your function will compute the given number of “steps” elements of the series.

The `pi_taylor_serial.cc` program skeleton already reads the number of steps and calls the `pi_taylor` function that you need to complete. Also note that the floating point type can be selected with the `my_float` defined type at the beginning of the program. The current `long double` float type increases numeric precision. Also, the program uses std::setprecision to print as many decimal digits as possible.

3.1.1 Question 1

What is the goal of the `std::stoll` function?

3.1.2 Question 2

Please think in any parallelizable approach for the pi_taylor program?

3.2 SYCL versions

Now, we are going to implement a parallel version of the same π approximation algorithm. The new `pi_taylor_sycl` program will have two arguments, the number of steps and the number of threads, and the first value has to be larger than the second.

The repo includes a pi_taylor_sycl.cc skeleton that can help you with the task. Could this program benefit from the `parallel_for` from the previous lab?

3.2.1 Question 1

Can you think in at least two possible implementations of the pi taylor required reduction?

4 Laboratory 3: Remove Loop-Carried Dependency

One useful design pattern for High Level Synthesis with FPGA is to help the compiler to infer a shift register for removing a loop-carried dependency. This optimization applies to any long latency operation, and it benefits from the spatial locality of FPGAs to remove *artificial* dependencies between iterations.

As a simple example, please assume that you want to multiply all the elements of an array.

4.1 SYCL naive version

Please write the code to multiply an array of 1024 elements and save the result in the first element of another array. For example, the first and second arrays can be named as `a` and `result`.

4.1.1 Question 1

What is the initiation interval of the code? And what is it establishing its value?

4.2 SYCL version

5 Laboratory 4: Pipes

Pipes are an essential mechanism to split kernel functionality into simpler kernels and mitigate the dynamic behaviour; e.g., non-contiguous memory accesses.

To play with pipes, we are going to split the array addition example from the first lab into 4 independent single-task kernels, all running in the same queue. The first and second kernels will read A and B vectors, and write them into two blocking pipes that will be the input of the third kernel. This kernel will actually perform the addition and write its result into another pipe that the last kernel will read to store the result into the C vector.

Since pipes are an Intel extension, please include `#include <CL/sycl/intel/fpga_extensions.hpp>` in your source file.

5.0.1 Optional

Use multiple files, one per kernel, to test the multiple files compilation flow.

6 Final remarks

Please do not hesitate to share you comments, ideas, found typos to my e-mail dario@unizar.es.