

A Translation Framework from RVC-CAL Dataow Programs to OpenCL/SYCL based Implementations - A Comprehensive Starter's Guide

Code Conversion from RVC-CAL to C++/SYCL or C++/OpenCL

Florian Krebs

January 2, 2019

1 Introduction

This framework converts dataflow networks specified in RVC-CAL to executable C++/SYCL or C++/OpenCL code. The Reconfigurable Video Coding-CAL Actor Language (RVC-CAL) is a programming language based on CAL (<https://ptolemy.berkeley.edu/projects/embedded/caltrop/language.html>). RVC-CAL is part of the MPEG video coding standard (ISO/IEC 23001-4 or MPEG-B pt. 4). RVC-CAL is used to specify actors and complete dataflow networks. The actors are specified with CAL and the networks are specified in XML format. This framework can convert those networks and actors to C++17. One of the most popular RVC-CAL compilers is the Open RVC-CAL Compiler ORCC (<http://orcc.sourceforge.net/>). This framework is able to convert the working sample RVC-CAL projects (<https://github.com/orcc/orc-apps>) provided by ORCC to executable code. For this reason, this framework adopts the way native include files are used by ORCC. But in comparison to ORCC, this framework not only tries to generate code that is parallelized with the usual factories of the operating system. Instead, actions are also executed multiple times in parallel either with the SYCL or the OpenCL framework. The intention of this framework is to accelerate the execution of dataflow networks specified in RVC-CAL by the use of SYCL or OpenCL compared to a pure C/C++. Additionally, with the help of this framework, the performance of SYCL, especially the ComputeCpp implementation, and the performance of OpenCL can be compared.

2 Code Repository and Dependencies

The code of this framework can be downloaded from GitHub(<https://github.com/KrebsFlorian/MasterThesis>). There are four different versions of the code generator framework. One version to generate Cpp code that uses SYCL to parallelize the execution of actions with a GUI and one without a GUI. The same for the code generator for Cpp code that uses OpenCL. To compile the Code, a C++17 compiler is required.

To build and run the application the following libraries and framework are necessary:

- RapidXML (<http://rapidxml.sourceforge.net/>): The header file `rapidxml.hpp` is used to parse the network files.

- Qt 5.11.1 (<https://www.qt.io/>) and ideally the Visual Studio 2017 Plugin: The Qt framework is used for the GUI. The Visual Studio Plugin is convenient to set up a Visual Studio project with all necessary Qt includes.
- ComputeCpp (<https://www.codeplay.com/products/computesuite/computecpp>) and ideally the Visual Studio 2015 Plugin: This framework is an implementation of SYCL. Again the Visual Studio Plugin is convenient to create a project with all necessary includes already set. There is also an open-source implementation that could be used as well. But the code was tested with the ComputeCpp version.
- OpenCL Installation, e.g. the Intel SDK and the corresponding Visual Studio Plugin

The following examples rely on Visual Studio 2017 and Visual Studio 2015. Unfortunately, the older version has to be used to get the ComputeCpp SYCL implementation running.

3 Building the Framework

This section describes how the framework can be built either with a GUI or without with the example of Visual Studio 2017 on Windows 10. But first, some issues with the RapidXML library are addressed. This might be relevant if the compilation of the framework fails.

3.1 RapidXML

The used library RapidXML, especially the `radpidxml.hpp` uses functions that are regarded as insecure. Thus, during compilation, some compilers will abort and complain about this. This abortion can be avoided by using the `_CRT_SECURE_NO_WARNINGS` compiler preprocessor flag. In Visual Studio the flag is set in the properties of the project. (Right-click on the project and select properties) Then, *C/C++* and there select *Preprocessor*. In the Preprocessor menu add the flag to *Preprocessor Definitions*. Afterwards, it should look like in figure 1. It is not

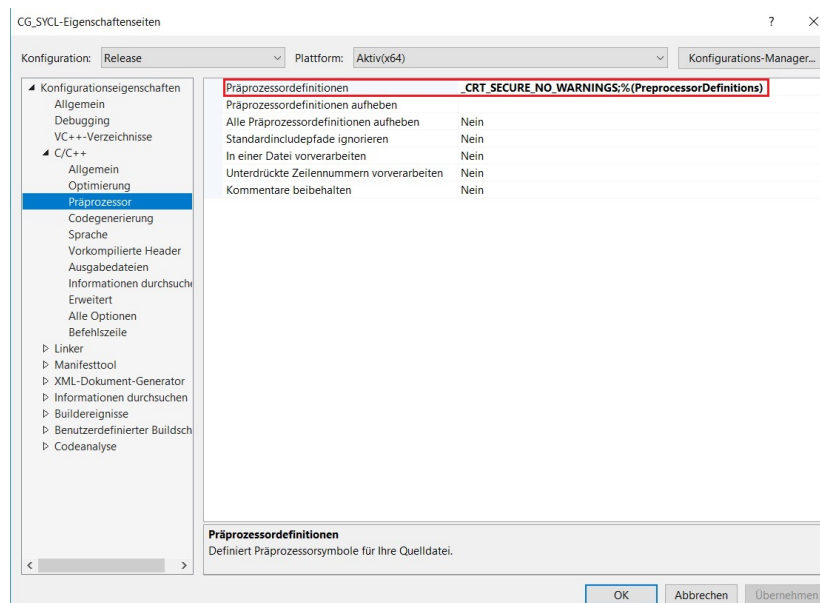


Figure 1: Add `_CRT_SECURE_NO_WARNINGS` to the Compiler Preprocessor Definitions

always necessary to add this flag to the Preprocessor Definitions. Some compilers won't care

about insecure functions. Thus, just add this flag if the compilation fails and the compiler complains about insecure functions or if your IDE shows a lot errors in the code. The IDE might treat insecure functions as errors without this flag and, therefore, show errors in other parts of the code.

3.2 Building the Framework with a GUI

To build the framework with a GUI Qt 5.11.1 or a compatible version is required. If the Qt Plugin for Visual Studio is installed, the project can be set up easily. To set up the project select the *Qt GUI Application* template as you can see in figure 2. Then follow the

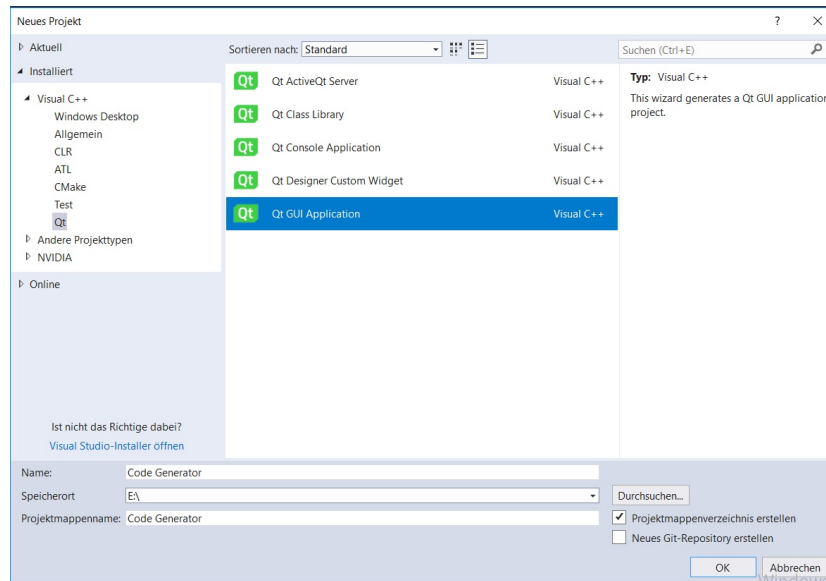


Figure 2: Choose the Qt GUI Application template

wizard without changing anything except in the third step. In the third step rename the Class Name either to *CodeGeneratorGUI* for the SYCL version or *CodeGeneratorOpenCLwithGUI* for the OpenCL version as you can see in figure 3. All the other file names are adjusted automatically by the wizard and require no further changes! After the creation of the project is finished, the source code has to be added to the project. The easiest way would be to copy all the files to the source folder of the project. Then all files that are not already included in the project by the template have to be included. Therefore, add all header files except *CodeGeneratorOpenCLwithGUI.hpp* or *CodeGeneratorGUI.hpp* to *Header Files* and the cpp files except *CodeGeneratorOpenCLwithGUI.cpp* or *CodeGeneratorGUI.cpp* to *Source Files*. (Right-click on the folder → Add → Add existing item) Please be aware that the GUI files are already added to the project. Thus, they are overwritten by copying the files into the project folder and no further action regarding the GUI files are necessary. If you have chosen not to copy the files to the project source folder and instead add them from a different location, the .ui file has to be replaced by the new one as well. Don't forget to add the RapidXML header *rapidxml.hpp* to *Header Files* as well. Afterwards, the project explorer should look similar to the project explorer in figure 4. The name of the project and the name of the GUI can vary depending on whether you are building the OpenCL or the SYCL Code Generator version. Due to the already included Qt library, the only step left is building the project. Now the framework can be executed either directly with the exe file or with Visual Studio. If the generated exe file shall be used, several Qt DLLs have to be copied

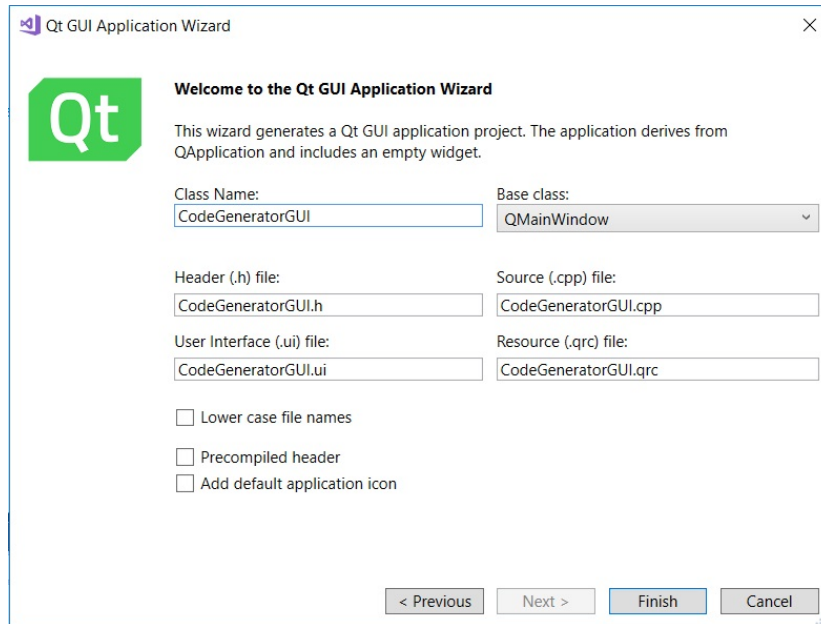


Figure 3: Adjust the name of the GUI classes in the Qt GUI Application wizard

to the folder where the exe file is located. Originally, the DLLs are located in *Qt folder/Qt version/msvc2017_64/bin*. The DLLs that have to be copied into the folder where the exe file is located are *Qt5Core.dll*, *Qt5Gui.dll* and *Qt5Widgets.dll*. Additionally, a new folder named *platforms* has to be created in the same folder and the *qwindows.dll* located in *Qt folder/Qt version/msvc2017_64/plugins/platforms* has to be copied into the new folder. Afterwards, it should look like in figure 5. The *windeployqt.exe* located in the same bin folder, where most of the necessary DLLs are, might also be useful. This tool can analyze Qt Applications and find all necessary DLLs and copy them to the folder where the analyzed executable is located. But this tool might find more DLLs than necessary. Now the executable should be working and usable.

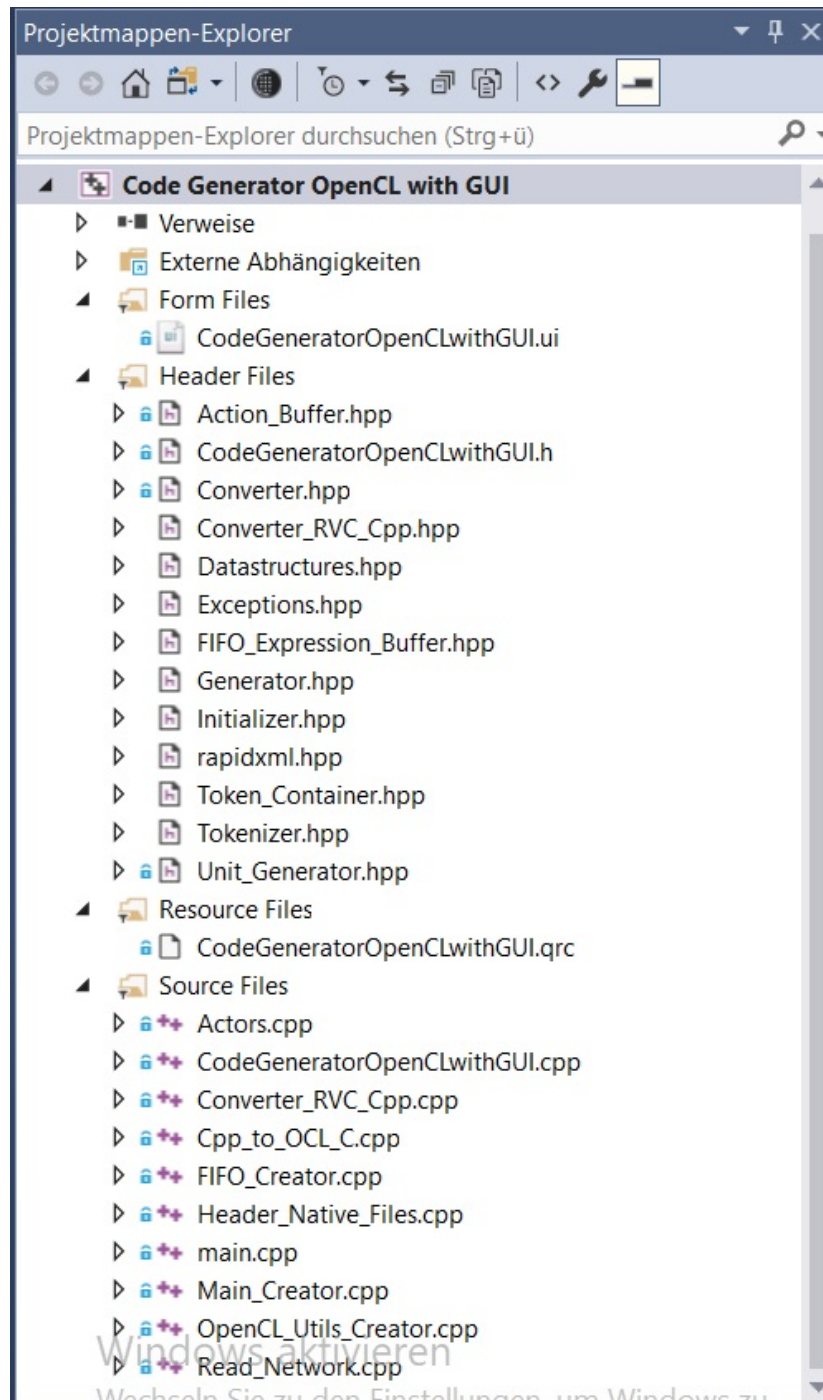


Figure 4: Visual Studio Project Explorer view after all the files have been added

repo > Synthese-von-OpenCL-aus-RVC-CAL > Implementierung > Code Generator with GUI > x64 > Release				
Name	Änderungsdatum	Typ	Größe	
platforms	07.10.2018 15:42	Dateiordner		
Code Generator with GUI.exe	02.10.2018 13:37	Anwendung	614 KB	
Qt5Core.dll	07.10.2018 15:42	Anwendungserwei...	5.778 KB	
Qt5Gui.dll	15.06.2018 07:13	Anwendungserwei...	6.190 KB	
Qt5Widgets.dll	15.06.2018 07:15	Anwendungserwei...	5.412 KB	

ese-von-OpenCL-aus-RVC-CAL > Implementierung > Code Generator with GUI > x64 > Release > platforms				
Name	Änderungsdatum	Typ	Größe	
qwindows.dll	15.06.2018 07:18	Anwendungserwei...	1.394 KB	

Figure 5: Executable with all necessary DLLs

3.3 Building the Framework without a GUI

To build the framework without a GUI, Qt isn't required. First, a new C++ Project has to be created. This project can be an empty one like in figure 6. After the project has been created,

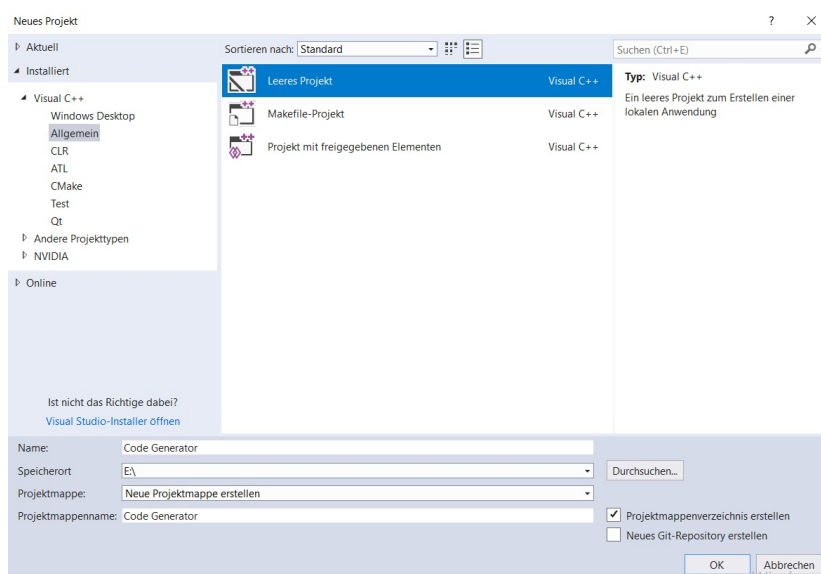


Figure 6: Create an empty C++ Project in Visual Studio

add all header files of the corresponding code generator version you want to build to *Header Files* or *Headerdateien* and all C++ files to *Source Files* or *Quelldateien*. (Right click on the folder → Add → Add existing item) Don't forget to add the RapidXML header *rapidxml.hpp* to *Header Files* or *Headerdateien*. If you have added all the files to the right location, your project explorer should look similar to the one in figure 7. Depending on whether you have chosen the C++/OpenCL version like in the figure or the C++/SYCL version, there might be slightly more or fewer files. But if in the project explorer all files are appearing like in the corresponding GitHub folder and the RapidXML header everything is fine. Now, you just need to build the project.

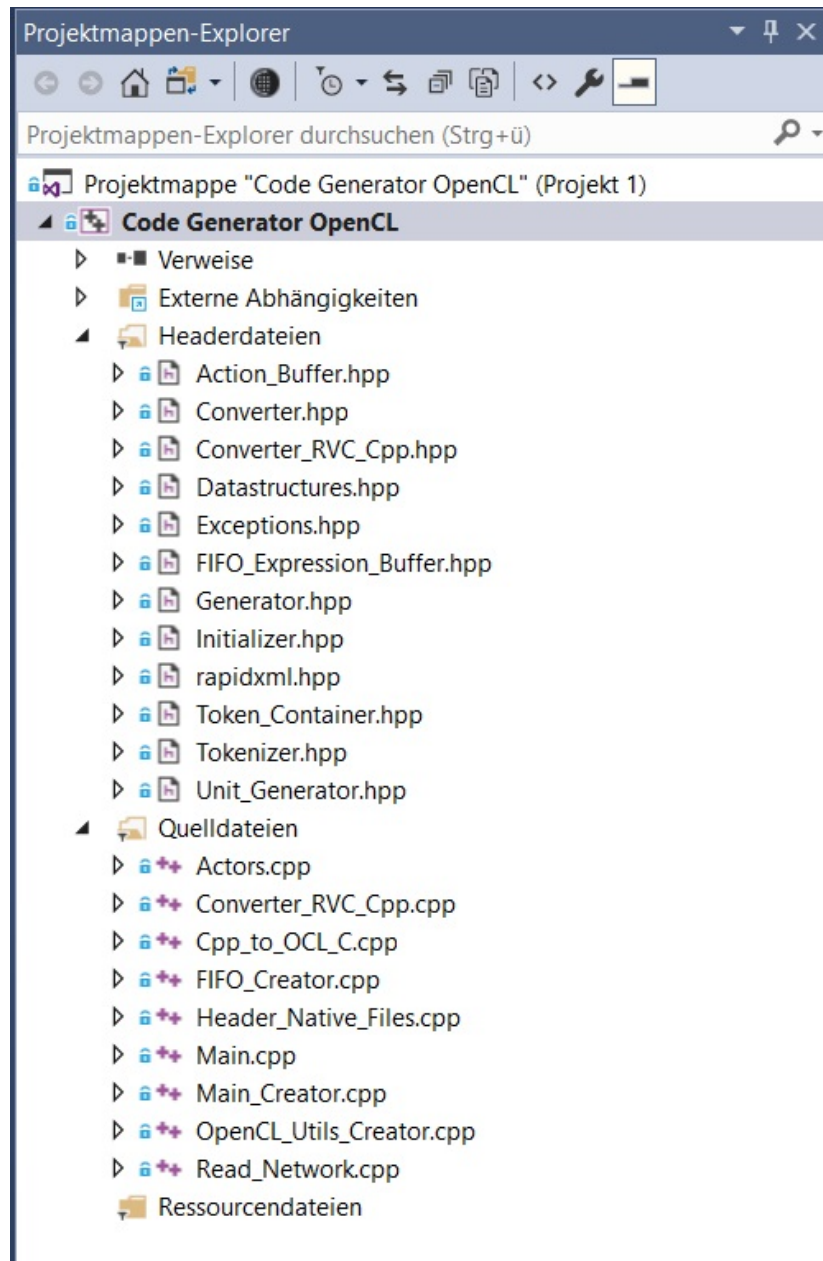


Figure 7: Visual Studio Project Explorer view of the C++/OpenCL Code Generator

4 Usage Scenarios of the Frameworks

In this section, the possible usage scenarios of the different framework versions are explained. Please note that the GUI versions can also be used in the same way than the versions without a GUI. All versions require that all actor files used in the network to be converted are located in the declared source folder. The code generator treats all paths to actor files used in the network as relative to the declared source folder. Thus, it might be necessary to copy some folders from other RVC-CAL projects, e.g. System, to the source folder of the RVC-CAL project to be converted to make conversion possible. This tool is just a code converter! It won't check the code for correctness in advance. Thus, if it is trying to convert erroneous code, the code generator might throw an exception, print something to the error output or

just get stuck in an infinite loop waiting for a token that won't appear. If something like this happens, please check the actor where the code generator crashed to see if some unsupported language construct has been used or if the actor is correct at all. A list and a brief explanation of all supported CAL Language features or statements can be found in the following section after the explanation of the usage scenarios.

Additionally, all versions are capable of producing pure C++ code that is not using SYCL or OpenCL. Code samples to test the code generator can be found in the GitHub repository of ORCC (<https://github.com/orcc/orc-apps>).

4.1 C++/SYCL Code Generation using the Command Line

To generate C++/SYCL code for a dataflow network specified in RVC-CAL, the network file, the source directory of the project and the output directory where the generated code files shall be placed have to be inputted into the code generator. For this purpose the code generator offers command line options:

- `-n < file >` : This option is used to specify the top network file that shall be converted to C++
- `-w < directory >` : This option is used to specify the output directory where the generated files are written to. This folder has to exist already, otherwise, the code isn't written somewhere.
- `-d < directory >` : This option is used to specify the source directory of the actors and networks related to the top network specified with the `-n` option.

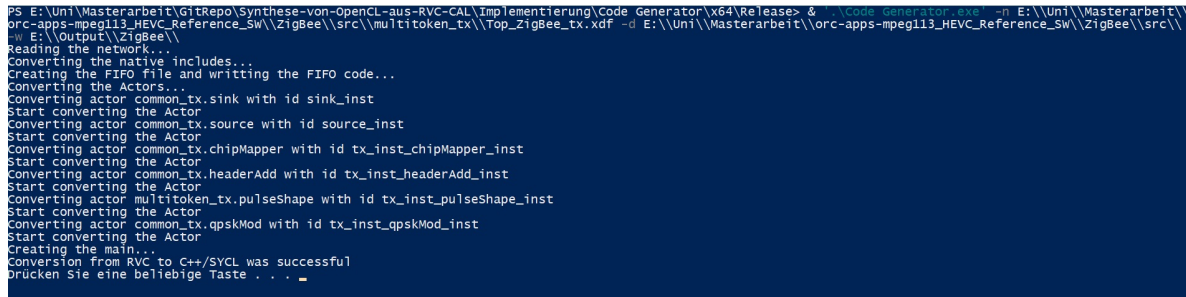
These three command line options are the most important options to use the most basic functionality of the code generator. But there are some more options that give the user more control over some properties of the generated code:

- `-s < number >` : This option gives the user the opportunity to specify the size of the FIFO buffers. By default, the size is set to 100000. To exploit parallelism, the size of the FIFO shouldn't be too small, because this would lead to less parallel instances of an action due to the lower number of available tokens at this moment.
- `-i < file >` : This option can be used to specify c files containing native functions. This option can be used multiple times to specify multiple c files. The code generator produces for every c file a header and sets the corresponding includes in the actor code.
- `-p` : If this flag is set, the code generator won't produce code that is using SYCL. Instead, it will produce pure C++ code.
- `-cpu` : The SYCL calls are executed on the CPU. By default, they are executed on a GPU.
- `-c < number >` : If this option is used, the target of the SYCL calls is the CPU. For performance reasons, in this case only as many workitems are used as many cores are available. The number of cores has to be specified after the `-c` option. This is just an experimental option, the performance gain is low, but it's there.

All those options can be displayed with the `-h` option. Unfortunately, the code generator is not checking includes and code of the C files that are inputted with the `-i` option for correctness. This option is only there for convenience. Otherwise, the header files must be

created completely manually and the includes also must be set by hand. In this case, the include statements have to be inserted into the extern "C" at the beginning of each actor code calling native functions. Thus, this feature can make things easier, but the generated header file should be checked for errors and unnecessary include-statements that should be removed.

An example of code generation can be seen in figure 8. Now, the generated code file are in



```
PS E:\Un\Masterarbeit\GitRepo\Synthese-von-OpenCL-aus-RVC-CAL\Implementierung\Code_Generator\Release> & .\code_generator.exe -n E:\Un\Masterarbeit\orc-apps-mpeg113_HEVC_Reference_Sw\ZigBee\src\multitoken_tx\Top_ZigBee_tx.xdf -d E:\Un\Masterarbeit\orc-apps-mpeg113_HEVC_Reference_Sw\ZigBee\src\ -w E:\Output\ZigBee\
Reading the network...
Converting the native includes...
Creating the FIFO file and writting the FIFO code...
Converting the Actors...
Converting actor common_tx.sink with id sink_inst
Start converting the Actor
Converting actor common_tx.source with id source_inst
Start converting the Actor
Converting actor common_tx.chipMapper with id tx_inst_chipMapper_inst
Start converting the Actor
Converting actor common_tx.headerAdd with id tx_inst_headerAdd_inst
Start converting the Actor
Converting actor multitoken_tx.pulseShape with id tx_inst_pulseShape_inst
Start converting the Actor
Converting actor common_tx.qpskMod with id tx_inst_qpskMod_inst
Start converting the Actor
Creating the main...
Conversion from RVC to C++/SYCL was successful
Drücken Sie eine beliebige Taste . . .
```

Figure 8: Code Generation for the ZigBee Project

the given output folder.

4.2 C++/SYCL Code Generation using the GUI

To use the GUI the generated executable has to be executed. Then the GUI is displayed. If the GUI is not starting and some Qt Error is displayed, please go back to the building step and check if all required DLLs are in the right location. These DLLs might vary from system to system and between different Qt versions. The GUI for the code generator for C++/SYCL code should look like in figure 9. Before the code generation can start, the Network File, Source Directory and Output Directory must be specified. Therefore, the ...-Button on the right of the corresponding text fields can be used. This button opens a file or directory selector to select the requested network, source and output directory. These three fields are absolutely mandatory for the code generator to work properly. If you choose to state them without the file and directory selectors, please make sure that the stated output folder exists. Otherwise, the files will be created in a not existing directory. This won't trigger an exception, but the files won't be written either. In the next part, paths to C files can be inputted. These C files should contain the functions associated to the used native functions in actors. It is just optional to state them because this functionality is just for the convenience of the user. This function generates a header file for the C file and sets the corresponding includes in the generated code where native functions are called. Otherwise, the user has to do this manually. In this case, the include statements have to be inserted into the extern "C" block at the beginning of each actor code calling native functions. But please be aware that this feature doesn't check the generated header file for errors and unnecessary imports. Finally, in the below right part the desired target for SYCL or no SYCL at all can be stated. The SYCL targets can be either the GPU, the CPU using as many workitems as there are cores or the CPU without taking the number of cores into consideration. Additionally, the number of cores can be inputted, but this is only relevant if *Target CPU with Core limitation* is selected. For performance reasons, only as many workitems as there are cores are used to execute the SYCL calls. To the right of the text field where the number of cores can be inputted, the size of the FIFO buffers can be specified. If this field is empty, the default 100000 is used. This number should not be selected too low, because this can limit the degree of parallelism. The number of available tokens in the input queues and the available free space in output queues

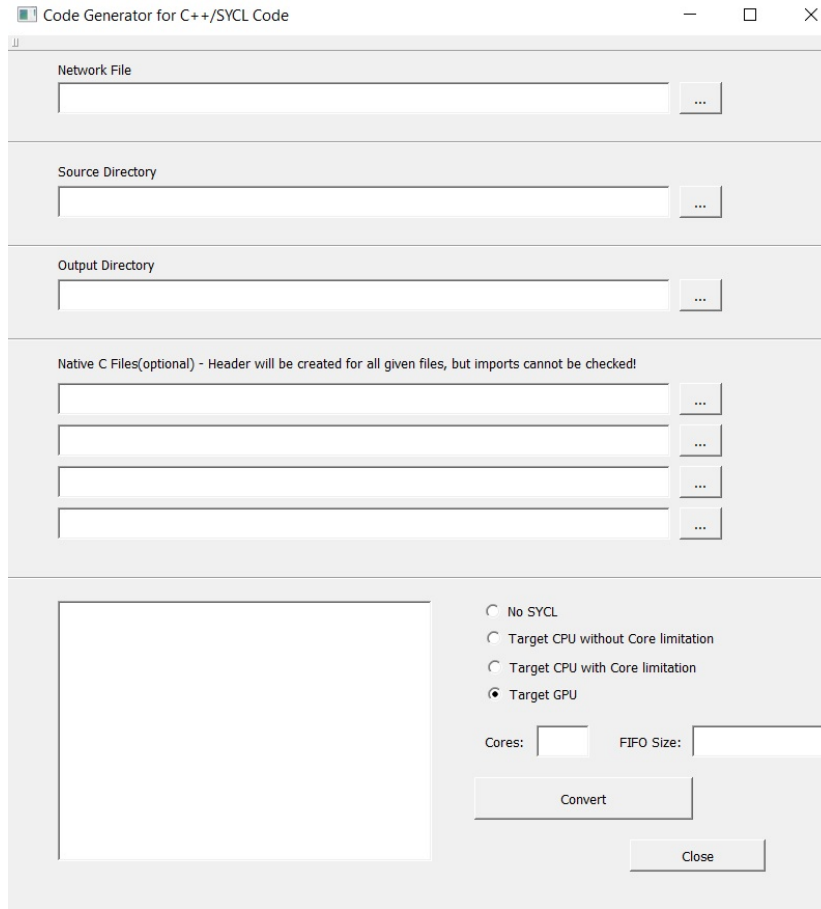


Figure 9: Graphical User Interface of the Code Generator for C++/SYCL Code

determine how many instances of an action are executed in parallel. Thus, numbers around 1000000 seem to be quite optimal.

After all the desired or requested values have been inputted, the code generation can be started with the *Convert* button. During conversion, status reports are displayed in the text field below left. When *Conversion from RVC to C++/SYCL was successful* is displayed, the code generation is completed and the code can be found in the given output folder.

4.3 Building and Executing the Generated C++/SYCL Code

After code generation, the final step is to build the executable. To set the SYCL dependencies, Visual Studio 2015 with the ComputeCpp Plugin is quite useful. Unfortunately, the plugin seems to be not working in Visual Studio 2017 by now. Thus, using Visual Studio 2015 seems to be the best way to get ComputeCpp running. If the ComputeCpp Visual Studio Plugin is installed, the *ComputeCpp SYCL C++* template can be used to create a new C++ project with all required dependencies, like shown in figure 10. Without this template, the SYCL and OpenCL dependencies have to be set by hand. The next step is to add all the generated files to the project. Therefore, again add the header files to *Header Files* and the source files to *Source Files*. (Right-click on the folder → Add → Add existing item) Afterwards, the project explorer using the example of ZigBee Multitoken should look like in figure 11. Before building this project, all other external dependencies required by the C Code must be added, e.g. SDL to display pictures on the screen. Afterwards, the project can be built. However, building

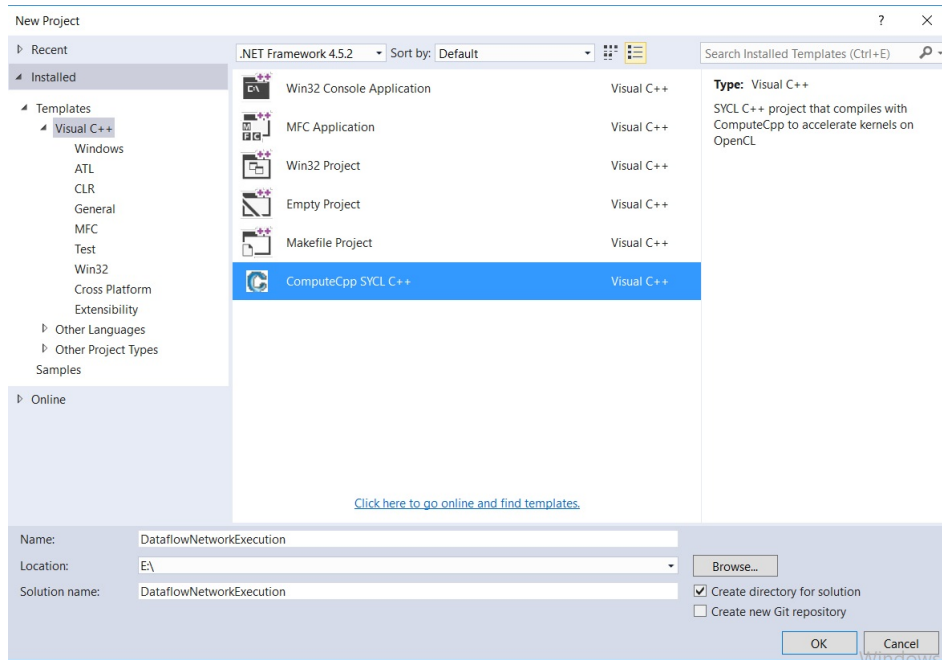


Figure 10: Select the *ComputeCpp SYCL C++* Template to create a C++ Project with SYCL Dependencies

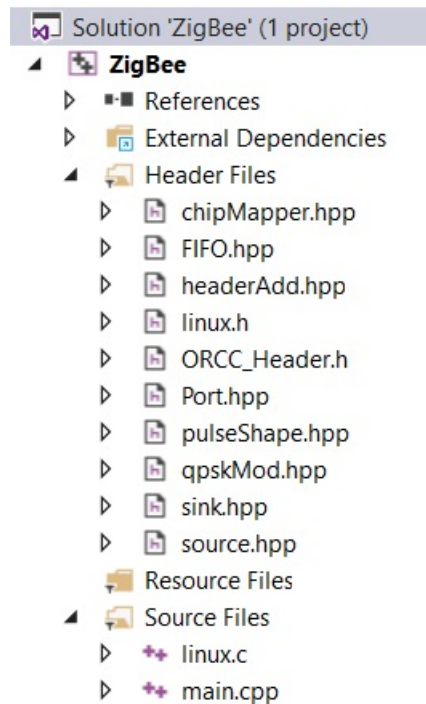


Figure 11: Visual Studio 2015 Project Explorer View using the Example of ZigBee Multitoken

might fail due to compiler settings. Unfortunately, SYCL seems to require an older compiler version. By default, this template sets the Target Platform Version to 8.1 and the Platform Toolset to Visual Studio 2015 (v140). This must remain the same! Any change will cause the building process to fail. Unfortunately, if the Platform Toolset wasn't changed the building

process could fail, too. This older version has some problems with native casting of values, especially when using initializer lists to initialize an array. Thus, it might be necessary to add some casts or replace auto classifiers by concrete types. But this can vary from compiler to compiler. The newest version of Visual++ builds the generated C++ code without any errors. The build process might also fail due to the stack size or too many arrays being allocated on the stack. The code generator allocates everything on the stack. If large arrays are used, the required memory might exceed the stack size. Thus, either the stack size has to be increased or these arrays have to be allocated on the free store or heap. If the second option is used, please pay attention to the fact, that the allocated memory potentially has to be freed again, this is true for all variables except class variables.

After the code has been built successfully, the executable can be used in pretty much the same way than an executable built from ORCC generated sources. The most relevant command line parameters are:

- `-i < file >` : This option is used to specify the input file.
- `-w < file >` : This option is used to specify the file where the output is written to.
- `-l < number >` : This option specifies how often the content of the input file is read. By default, the input file is read once.
- `-f < number >` : This option specifies how many frames will be decoded before exiting.

But not all of these options are used by all native files. For example, the number of decoded frames doesn't make sense in the context of ZigBee and, therefore, won't be used in this context. All options are displayed with the help of the `-h` option. The previously built ZigBee Multitoken application can be launched with the following command: `./ZigBee.exe -i ./tx_stream.in -w ./output.out` The file used here with test input data, `tx_stream.in`, can be found in the `ZigBee/lib/input_signals` folder of the sample project repository of ORCC.

4.4 C++/OpenCL Code Generation using the Command Line

To convert a dataflow network specified in RVC-CAL to C++/OpenCL Code, the network file, the source directory of the project and the output directory where the generated code files shall be placed have to be inputted to the code generator. For this purpose the code generator offers command line options:

- `-n < file >` : This option is used to specify the top network file that shall be converted to C++
- `-w < directory >` : This option is used to specify the output directory where the generated files are written to. This folder has to exist already, otherwise, the code isn't written somewhere.
- `-d < directory >` : This option is used to specify the source directory of the actors and networks related to the top network specified with the `-n` option.

These three command line options are the most important options to use the most basic functionality of the code generator. But there are some more options that give the user more control over some properties of the generated code:

- `-s < number >` : This option gives the user the opportunity to specify the size of the FIFO buffers. By default, the size is set to 100000. To exploit parallelism, the size of the FIFO shouldn't be too small because this would lead to less parallel instances of an action due to the lower number of available tokens at this moment.
- `-i < file >` : This option can be used to specify c files containing native functions. This option can be used multiple times to specify multiple c files. The code generator produces for every c file a header and sets the corresponding includes in the actor code.
- `-p` : If this flag is set, the code generator won't produce code that is using OpenCL. Instead, it will produce pure C++ code.

All those options can be displayed with the `-h` option. Unfortunately, the code generator is not checking includes and code of the C files that are stated with the `-i` option. This option is only there for convenience. Otherwise, the header files must be created completely manually and the includes also must be set by hand. In this case, the include statements have to be inserted into the extern "C" at the beginning of each generated actor code calling native functions. Thus, this feature can make things easier, but the generated header file should be checked for errors and unnecessary include-statements that should be removed.

An example of code generation can be seen in figure 12. If *Conversion from RVC to*

```
PS E:\Uni\Masterarbeit\GitRepo\Synthese-von-OpenCL-aus-RVC-CAL\Implementierung\Code Generator\OpenCL\x64\Release> & "C:\Program Files\Code Generator\OpenCL.exe" -n E:\Uni\Masterarbeit\orc-apps-mpeg13_HEVC_Reference_Sw\ZigBee\src\multitoken_tx\Top_ZigBee_tx.xdf -d E:\Uni\Masterarbeit\orc-apps-mpeg13_HEVC_Reference_Sw\ZigBee\src -w E:\Output\ZigBee -i E:\Uni\Masterarbeit\orc-apps-mpeg13_HEVC_Reference_Sw\ZigBee\lib\native\linux.c
Reading the network...
Converting the native includes...
Creating the FIFO file and writing the FIFO code...
Creating the OpenCL utilities...
Converting the Actors...
Converting actor common_tx.sink with id sink_inst
Converting actor common_tx.source with id source_inst
Converting actor common_tx.chipMapper with id tx_inst_chipMapper_inst
Converting actor common_tx.headerAdd with id tx_inst_headerAdd_inst
Converting actor multitoken_tx.pulseShape with id tx_inst_pulseShape_inst
Converting actor common_tx.qpskMod with id tx_inst_qpskMod_inst
Creating the main...
Conversion from RVC to C++/OpenCL was successful
Drucken Sie eine beliebige Taste . . .
```

Figure 12: C++/OpenCL Code Generation for the ZigBee Project

C++/OpenCL was successful is displayed, the code generation was successful. Now, the generated code can be found in the given output folder.

4.5 C++/OpenCL Code Generation using the GUI

To use the GUI the generated executable has to be executed. Then the GUI is displayed. If the GUI is not starting and some Qt Error is displayed, please go back to the building step and check whether all required DLLs are in the right location. These DLLs might vary from system to system and between different Qt versions. The GUI for the code generator for C++/OpenCL code can be seen in figure 9. Before code generation, the Network File, Source

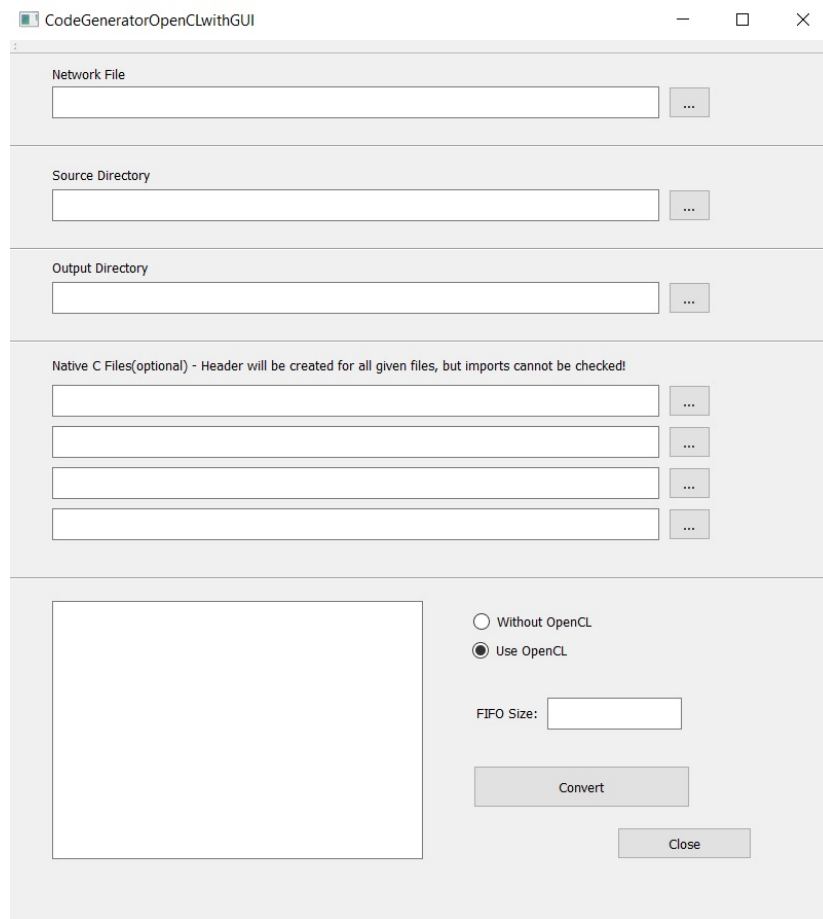


Figure 13: Graphical User Interface of the Code Generator for C++/OpenCL Code

Directory and Output Directory has to be specified. Therefore, the ...-Button on the right of the corresponding text fields can be used. This button opens a file or directory selector to select the requested network, source and output directory. These three fields are absolutely mandatory for the code generator to work properly. If you choose to input them without the file and directory selectors, please make sure that the given output folder exists. Otherwise, the files will be created in a not existing directory. This won't trigger an exception, but the files won't be written either. In the next part, C file paths can be inputted. These C files should contain the used native functions. It is just optional to specify them because this functionality is just for the convenience of the user. This function generates a corresponding header file to the given C file and sets the corresponding includes in the generated actor's code calling native functions. Otherwise, this has to be done manually. In this case, the include statements have to be inserted into the extern "C" block at the beginning of each actor code calling native functions. But please be aware that this feature doesn't check the generated header files for errors and unnecessary imports. Finally, in the below right part, it can be

selected whether OpenCL based code or pure C++ code shall be generated. Additionally, below there is a text field where the size of the FIFO buffers can be specified. If this field is empty, the default value 100000 is used. This number should not be selected too low, because this can limit the degree of parallelism. The number of available tokens in the input queues and the available free space in output queues determines how many instances of an action be executed in parallel. Thus, numbers around 1000000 seem to be quite optimal.

After all the desired or requested values have been specified, the code generation can be started with the *Convert* button. During conversion, status reports are displayed in the text field below left. When *Conversion from RVC to C++/OpenCL was successful* is displayed the code generation is completed and the code can be found in the given output folder.

4.6 Building and Executing the Generated C++/OpenCL Code

After code generation, the final step is to build the executable. To set all the OpenCL dependencies the Intel OpenCL Visual Studio plugin is quite useful. If the Intel OpenCL Visual Studio Plugin is installed, the *Empty OpenCL Project for Windows* template can be used to create a new C++ project with all required dependencies, like shown in figure 14. This

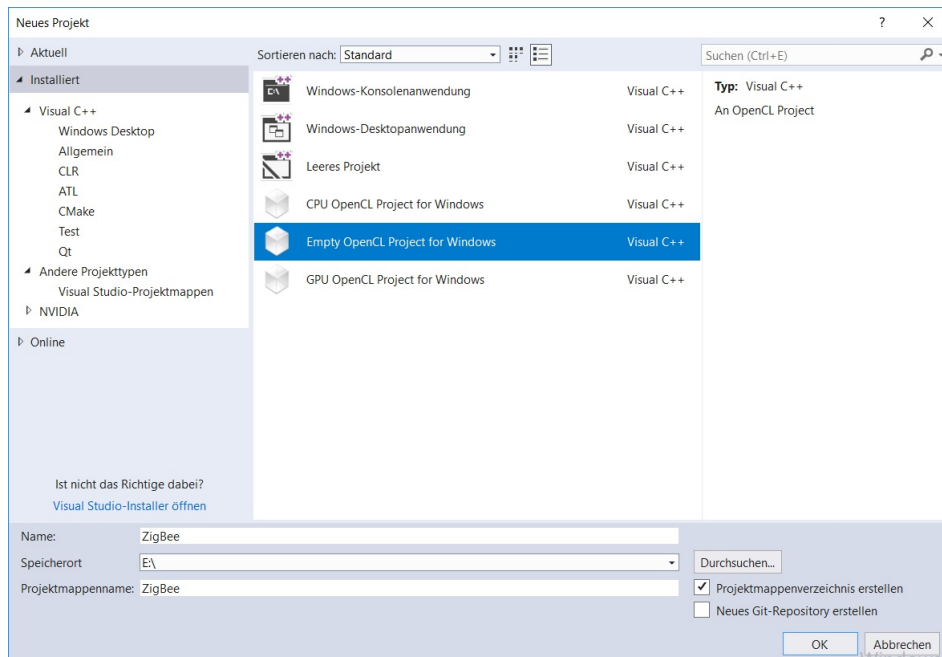


Figure 14: Select the *Empty OpenCL Project for Windows* Template to create a C++ Project with OpenCL Dependencies

seems to be the easiest way to set the OpenCL dependencies in Visual Studio 2017. Otherwise, the dependencies have to be set by hand. The next step is to add all the generated files to the project. Therefore, again add the header files to *Header Files*, the source files to *Source Files* and the cl files to *OpenCL Files*. (Right-click on the folder → Add → Add existing item) Afterwards, the project explorer using the example of ZigBee Multitoken should look like in figure 15. OpenCL doesn't automate the device selection, like SYCL does. Thus, the device manufacturer name and the type of the device has to be set manually in every actor class that is using OpenCL. Using the example of ZigBee Multitoken, the actors qpskMod and chipMapper are using OpenCL. To set the device where the OpenCL calls of an actor are executed on, both parameters mentioned before have to be adjusted in the constructor like

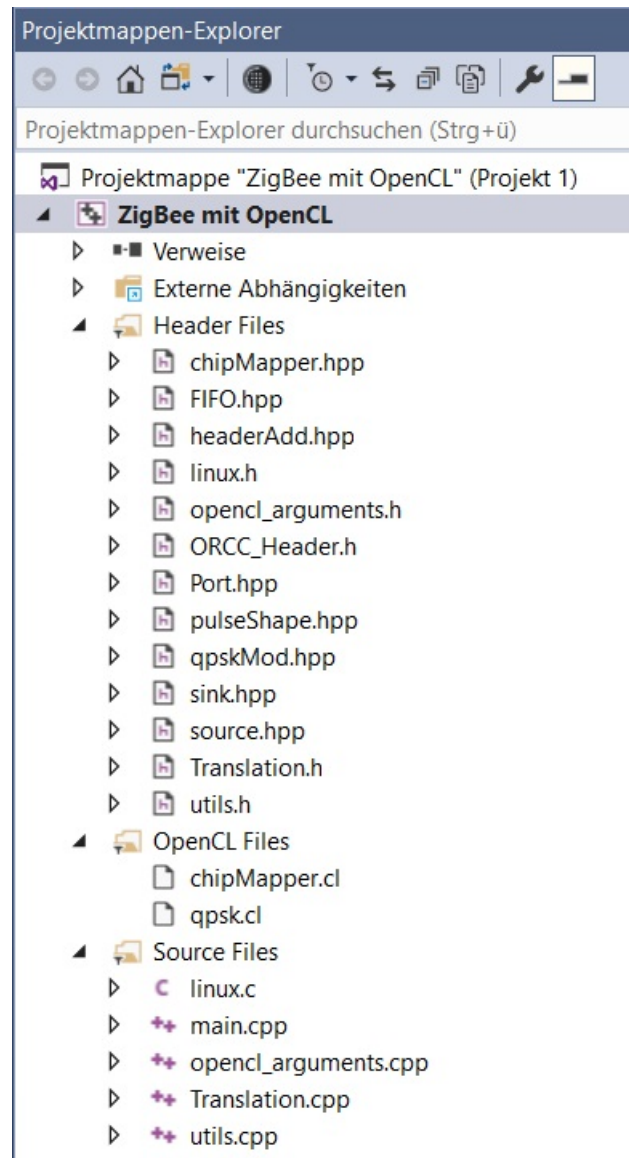


Figure 15: Visual Studio 2017 Project Explorer View using the Example of ZigBee Multitoken

shown in figure 16. In the figure, the default setting is shown, but it might not be useful to execute the code on an NVIDIA GPU or there even might not be one at all. Additionally, the OpenCL memory flag can be adjusted to the hardware. By default, the OpenCL buffers are created only with the memory flag `CL_MEM_READ_WRITE`, like it is shown in figure 17. The memory flags can be adjusted in the class `Port` in methods `get_read_buffer` and `get_write_buffer` in the file `Port.hpp`. Thus, those two parameters must be adjusted to the kernel requirements and the available devices. For example, `CL_DEVICE_TYPE_CPU` and "Intel" could be another option to execute the kernel on an Intel CPU. But please be aware that different devices have different requirements for kernels, e.g. the memory size of the device or the number of accepted kernel arguments could vary. Before building this project, all other external dependencies required by the C Code must be added, e.g. SDL to display pictures on the screen. Afterwards, the project can be built. The build process might fail due to the stack size or too many arrays being allocated on the stack. The code generator allocates everything on the stack. If large arrays are used, the required memory might exceed

```

    }
    chip$FIFO.openc1_read_done();
    symb$FIFO.openc1_write_done(ocl);
}
public:
qpskMod(Port< unsigned int , 100000>& _chip$FIFO,Port< char , 100000>& _symb$FIFO) : chip$FIFO(_chip$FIFO),symb$FIFO(_symb$FIFO) {
    cl_int err;
    if (CL_SUCCESS != SetupOpenCL(&ocl, CL_DEVICE_TYPE_GPU, "NVIDIA")) {
        exit(-1);
    }
    if (CL_SUCCESS != CreateAndBuildProgram(&ocl, "qpsk.cl"))
    {
        exit(-1);
    }
    ocl.kernel = clCreateKernel(ocl.program, "action$2", &err);
    if (CL_SUCCESS != err)
    {
        printf("Error: clCreateKernel returned %s\n", TranslateOpenCLError(err));
        exit(-1);
    }
}

void schedule(){
    for(;;){
        if(chip$FIFO.get_size() >= 1){
            if(symb$FIFO.get_free_space() >= 32) {
                action$2();
            }
        }
        else {

```

Figure 16: Adjust Device Selection Parameters

```

template<typename T, int N>
cl_mem* Port<T, N>::get_read_buffer(const int number_elements, openc1_arguments &ocl) {
    cl_int err = CL_SUCCESS;
    read_buffer = clCreateBuffer(ocl.context, CL_MEM_READ_WRITE /*| CL_MEM_ALLOC_HOST_PTR*/, sizeof(T) * number_elements, NULL, &err);
    if (CL_SUCCESS != err)
    {
        printf("Error: clCreateBuffer for inputBuffer returned %s\n", TranslateOpenCLError(err));
        exit(err);
    }
    T *resultPtr = (T *)clEnqueueMapBuffer(ocl.commandQueue, read_buffer, true, CL_MAP_WRITE, 0, sizeof(T) * number_elements, 0, NULL, NULL, &err);
    (...)
    return &read_buffer;
}

template<typename T, int N>
cl_mem* Port<T, N>::get_write_buffer(const int number_elements, openc1_arguments &ocl) {
    cl_int err = CL_SUCCESS;
    write_buffer = clCreateBuffer(ocl.context, CL_MEM_READ_WRITE /*| CL_MEM_ALLOC_HOST_PTR*/, sizeof(T) * number_elements, NULL, &err);
    if (CL_SUCCESS != err)
    {
        printf("Error: clCreateBuffer for outputBuffer returned %s\n", TranslateOpenCLError(err));
        exit(err);
    }
    write_offset = number_elements;
    return &write_buffer;
}

```

Figure 17: Adjust OpenCL Memory Flags in the Class Port

the stack size. Thus, either the stack size has to be increased or these arrays have to be allocated on the free store or heap. If the second option is used, please pay attention to the fact, that the allocated memory potentially has to be freed again, this is true for all variables except class variables.

Additionally, the build process might fail due to errors in the OpenCL code. Some built-in functions to OpenCL like min, max might be redefined in the code. To fix this issue either the function and all calls have to be renamed or the definition has to be removed. The second option should only be used if the redefined function and the built-in function have the same semantics. The build process can also fail for a similar reason. It is also possible that variables are named as one of the built-in functions or any other keyword of OpenCL C. Thus, they have to be renamed. The code generator checks for some of the popular keywords, like min, max or constant. But there are many more in OpenCL and, therefore, it is possible that the code generator won't rename one of them.

After the code has been built successfully, the executable can be used in pretty much the same way than an executable built from ORCC generated sources. The most relevant command line parameters are:

- `-i < file >` : This option is used to specify the input file.
- `-w < file >` : This option is used to specify the file where the output is written to.
- `-l < number >` : This option specifies how often the content of the input file is read. By default, the input file is read once.
- `-f < number >` : This option specifies how many frames will be decoded before exiting.

But not all of these options are used by all native files. For example, the number of decoded frames doesn't make sense in the context of ZigBee and, therefore, won't be used in this context. All options are displayed with the help of the `-h` option. The previously built ZigBee Multitoken application can be launched with the following command: `./ZigBee.exe -i ./tx_stream.in -w ./output.out` The file used here with test input data, `tx_stream.in`, can be found in the `ZigBee/lib/input_signals` folder of the sample project repository of ORCC.

5 Supported CAL Language Features

This section summarizes all language constructs that are supported by the code generator. This list can be used to check whether code contains unsupported language constructs and, therefore, code won't be generated or only erroneous code is generated because things might be interpreted wrong by the code generator. The supported CAL and RVC subset is:

- Actors: Including parameters and their default values. But list comprehension isn't supported in this part. It's also supported that the size of the parameter types or FIFO types is specified by a constant that is defined in the actors' body or in an included unit file.
- Actions: Including list comprehension, list assignment and inline if in the output FIFO part.
- Procedures
- Functions
- if ... then ... else ... end: Please be aware of that elseif, elif and every similar expression are not supported. But this behaviour can be achieved with a nested if statement in the else part. But this if must also be closed by an end, like the outer if-else!
- inline if ... then ... else .. end: Unless the inline if doesn't contain a list comprehension or list assignment, it is converted to a ternary conditional operator. Otherwise, it is converted a regular if-then-else.
- Finite State Machines
- priorities
- foreach-Loops
- while-loops
- Datatypes: (Unsigned) Integer, String, Float, Bool and Half including different sizes
- Lists: Fixed sized lists are supported. Even multi-dimensional lists are supported.
- List Comprehension: List comprehension is supported for actor variables and constants, FIFO outputs of actions, and variables and constants in functions, procedures, and actions. But list comprehension doesn't exist in C++. Thus, the array(lists are converted to arrays because they have a fixed size) is filled by a for-loop and it cannot be constant anymore. Hence, constant lists that are filled with list comprehension are not constant anymore after the conversion to C++ code.
- Arrays: Arrays are not part of CAL, but they are widely used. Thus, this code generator supports arrays in the same way it supports lists.
- Units: The imported code or variables are copied into the code of the actor. Thus, if there is a chain or tree of imports, multiple declarations of the same symbol could be in the generated code. Then there are unit files that are imported multiple times. Hence, the imports should be optimized.
- Native Function Imports: Only C Code is supported! Every actor calling native functions includes the header for the C files in an extern "C" block.

- Build-in Functions: min, max, print, println: print and println are converted to `std::cout` and min and max to `std::min` and `std::max`.
- Arithmetic and Logic Operations equivalent to C++: All arithmetic and logic operations are taken over to C++ Code like they are in RVC-CAL Code. There might be operations that are implemented slightly different in RVC-CAL than in C++. This could cause problems because it could change the semantics of the program.