University of Glasgow

School of Computing Science

Transformation of pipe-based OpenCL kernels into an LLVM-based intermediate representation for FPGA programming

XiaoWang, Songpei Xu, Jing Pan, Sijia Niu

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

< 23/12/2018 >

Abstract

FPGAs are excellent accelerators with wide use when dealing with the dataflow computation. The FPGAs' programming can be based on the OpenCL framework, with pipes to support dataflow programming. The OpenCL frameworks are being complemented by optimizing transformations in "*TyTra*" project. These transformatios require translation from pipe-based OpenCL kernels, to an LLVM-based custom intermediate representation called "*TyTra-IR*" . Hence, we designed a flow to complete these transformations, which uses generates *TyTra-IR* language via LLVM. In this report, we will introduce the general backgrounds of *TyTra* project, LLVM, OpenCL, and FPGA, and present the main process of our flow. After that, we present our source to source compiler which parses LLVM-IR and generates *TyTra-IR*. We have validated the effectiveness of the flow we designed, as we show our generated TyTra-IR successfully compiled by the *TyTra* back-end to generate FPGA code, which makes it easier to deploy scientific code onto FPGAs.

Key Words: FPGA, OpenCL, LLVM, *TyTra-IR*

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Jing Pan, Songpei Xu, Xiao Wang, Sijia Niu

Signature: Jing Pan, Songpei Xu, Xiao Wang, Sijia Niu

Acknowledgements

First of all, we would like to extend our sincere gratitude to our supervisor, Wim Vanderbauwhede, for giving us the general idea of the whole project which help us to know what we are going to do and giving us the detailed instruction of the methods of the project, which help us complete the project. In addition, Wim has offered valued comments on the first draft of our report.

Secondly, we are desire to express our highest gratitude to Syed Waqar Nabi, who has spent a lot of time and patiently helped us with the technical problem of the project, and we did learn a lot thing from him, such as the use of LLVM and Clang, and how to write the python source-source translator.

Contents

Chapter 1 Introduction

Modern computing systems are becoming increasingly diverse. By providing large numbers of processor cores, most computing platforms now have great potential for performing many computations in parallel. For the optimisition of the performance, the use of increased numbers of heterogeneous computational cores as well as effective exploitation by software is being resorted to. Computer systems consisting of various platforms have great potential for performing tasks quickly and efficiently.

However, programming such heterogeneous computational systems is a great challenge. The specific challenge that need to be addressed is how to exploit the parallelism of a given computing platform, e.g. a multicore CPU, a graphics processor (GPU) or a Field-Programmable Gate Array (FPGA), in the best possible way, without having to change the original program. These different platforms have very different properties in terms of the available parallelism, depending on the nature and organisation of the processing cores and the memory. Although with great potential for parallelism, FPGAs have failed to perform on the high-level programming platforms like OpenCL due to the radically different architecture from CPUs and GPUs [Nabi, S. W. et al, 2017]. Thus, the key for FPGAs is to create the pipelines based high-level compiler for acceleration of its scientific applications.

In this paper, we describe a programming flow which can transform the pipe-based OpenCL kernels in to *TyTra-IR* programs with a series script such as LLVM-IR. This *TyTra-IR* is based on LLVM can generate the FPGA solution via a *TyTra* back end. Especially, we generate a python source-source translator, which has high performance to translate LLVM-IR into *TyTra-IR*. We compile the python code based on a simple sample, and we proposed three debug cases to validate the correctness and effectiveness of our project, which are proved that our project is strong and precise. After that, we use the code on the transformation from LLVM code to tirl file, which can be used by FPGA directly.

The rest of this paper is organized as follows. Chapter 2 presents related background surveys, including *TyTra* project, LLVM and Clang, OpenCL, FPGA. Chapter 3 presents the requirement of the project. Then, chapter 4 introduces every process on details by giving step description and graphs. The results from a giving OpenCL file as well as the debug results to prove that our python code can be used on general type of functions in OpenCL is discussed in Chapter 5. Next, chapter 6 concludes and describes the future work. Finally, chapter 7 states that who did what in both the final product and submitted report.

Chapter 2 Background

This project is connected to the *TyTra* research project, which aims to transform the pipe-based OpenCL kernels into a custom LLVM-based intermediate representation for FPGA programming called *TyTra-IR*. Hence, we will first discuss the researches about *TyTra* project firstly to have a general conception of the project we would finish, which helped us to allocate the work. Then we will present the conception of LLVM which is important to generate the LLVM intermediate representation. We also will introduce two basic conception of OpenCL and FPGA, which are the basis of our project.

2.1 Tytra project

The Tytra project, according to Nabi and Vanderbauwhede (2017), is a project that uses Tytra flow. The Tytra flow is a synthetic flow, which is based on typed-based program transformations.

According to Nabi and Vanderbauwhede (2017), the main principle behind the *TyTra* project is that the project uses type transformations to generate program variant and translate then to the *TyTra* intermediate language (IR) by tools such as the python source-source translator we used. Then the compiler analyses each *TyTra-IR* variant to choose one selected variant to move to a hardware description language (HDL). Finally, the solution will be generated by conducting the HLS framework integration process.

2.2 LLVM and Clang

LLVM is a framework of compilers, which is written for C++. LLVM can provide the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform (Lattner, 2004). The main function of the LLVM is that it can be used as a back-end for multiple programming language, and it also can provide optimizing and generate the code automatically for CPUs.

Belong with that, the LLVM contains a lot of sub-projects, the most popular one is the Clang. Clang is a C language family (which includes C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) compiler front end for LLVM project (clang.llvm.org). The aim of Clang is to become the unified parser for C-based languages, for example the conformance of C language and the compatibility of GCC (Lattner, 2008). Moreover, the Clang is several times faster than GCC, and it is fully BSD licensed, with much better end-user features such as warnings and errors.

In this project, we used the LLVM and Clang to deal with the giving OpenCL file to generate the LLVM-IR, which was used to create the Tytra file via parsing by python script.

2.3 OpenCL

OpenCL (Open Computing Language) is a framework that allow users to write programs of which the execution involves heterogeneous platforms of processors and hardware accelerators such as CPUs (Central Processing Units), GPUs (Graphics Processing Units), DSPs (Digital Signal Processors) and FPGAs (Field-Programmable Gate Arrays) (Stone & Shi, 2010).

In this paper, OpenCL will be used on FPGA, and the advantages of achieving OpenCL on FPGA is that OpenCL can reduce the requirement of this development. Specifically, traditional underlying language design needs engineers to balance the resource, area and parallelism etc., since the constraints imposed by the external interface must be considered, the timing closure problem needs to be addressed, but OpenCL can do these series steps, designers can use this on a new FPGA (Czajkowski et al., 2012), which means that this OpenCL code on FPGAs has better performance than use FPGAs directly.

When compiling OpenCL for FPGAs, it can add logic on functions, and connect them and implement the kernel with special functions. There are two methods can implement OpenCL on FPGAs (Shagrithaya et al., 2013), one is an auxiliary accelerator. The software is implemented in the CPU, then the FPGA is used to accelerate the operation of some modules, and the CPU and FPGA are connected by PCIe. The other is the SOC method. The CPU is embedded in the FPGAs, and this way can reduce the communication delay.

2.4 FPGA

FPGA (Field Programmable Gate Arrays) is a kind of semiconductor, which contains a collection of configurable logic blocks (Moore, A, 2017). It enables users to program customized digital logic in the field in a simpler and efficient way compared with other ways of building hardware. FPGAs are very flexible that it can be adapted or reprogrammed functionality requirements even after manufacturing. To perform certain tasks, FPGA exploits thousands even millions logic gates based on the Boolean algebra, for example, conjunction (AND), disjunction (OR) and negation (Not) operation. By using Boolean algebra, the negative or positive electrical pulses can be represented 0s and 1s to compute.

FPGAs have evolved from the first version which mainly used as hardware design in the mid-1980s to the programmable hardware realm at present. Recent years, FPGAs have become the mainstream choice in the high-performance computing (HPC) and big data (Nabi, et al, 2017 & Czajkowski, et al, 2012). However, with OpenCL based programming platforms, because the fundamental different architecture of FPGAs and its lower frequency compared with CPU and GPU, the heterogeneous parallel ramming language cannot be used in the same way with CPU or GPU. Thus, in order to make the FPGAs works together with high-performance computing devices like CPU or GPU, it is key to exploit custom, deep pipelines for the given kernels (Nabi, et al, 2017).

Chapter 3 Requirements

1. Transform the OpenCL code into C code that works in clang, this involves defining of macros to remove or change unsupported keywords and creating OpenCL API function stubs.

2. Transform this C code into LLVM IR, via the script provided in the repo.

3. Transform the LLVM IR into *TyTra-IR* via a custom Python script.

4. Validate correctness of the toolchain by comparison with a manual *TyTra-IR* reference.

Chapter 4 Design and Implementation

In this section we will discuss the design and implementation of our project. An overview of the flow is shown in Fig.1:

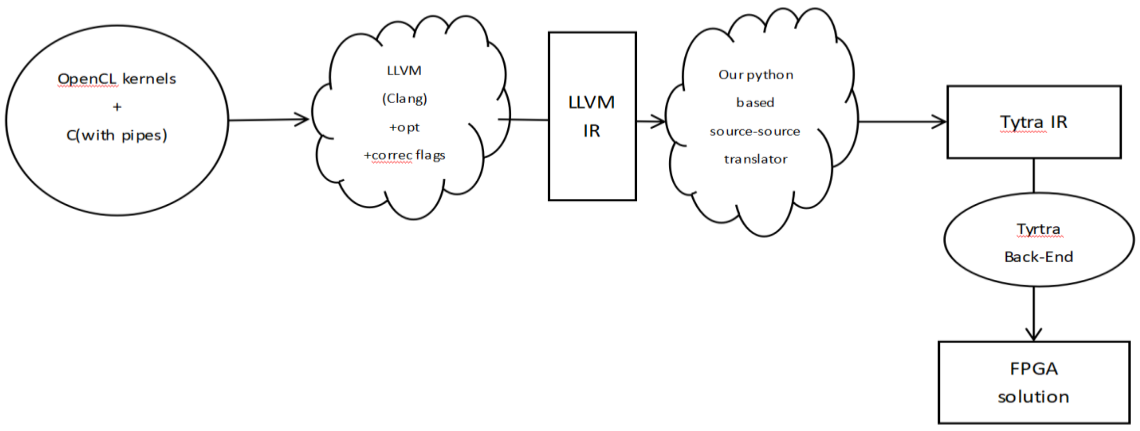


Fig.1. The design flow of the project

4.1 Write the OpenCL code with kernels and pipes.

The OpenCL-FPGA code that uses pipes typically has three kinds of kernels, including the input kernel, compute kernel and output kernel, and each kernel type may include several kernels. The data are read from global memory to input kernel and passed into compute kernels to do a series of operations. After that, the results are passed to output kernel and returned to global memory. The process is shown in Fig.2.

In order to complete this process, the OpenCL code has to be transformed to a type of code which can be read by the C front-end compiler (Clang), therefore, the original OpenCL code should define macros and create function stubs. Then the code can be compiled and generate the LLVM IR file. There are some limitations in our input OpenCL code, however, specifically, all input kernel are "scalarized", which means that array accesses are replaced by scalars, in addition, there is no loops in our input loops. On the other words, our test code just includes some simple operation. Nevertheless, this limitation is hardly influence our further work, because our further python code is based on a more general situation.

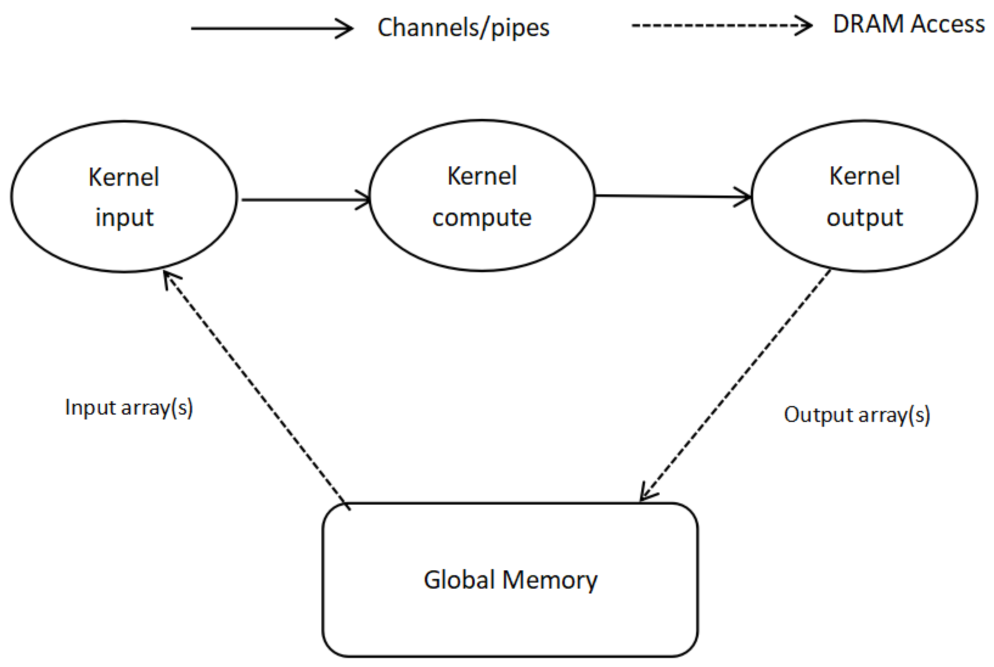
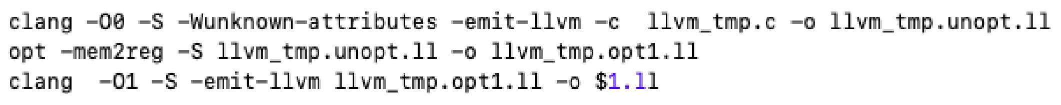


Fig.2. The process of OpenCL file

4.2 Transform the C code into LLVM IR.

LLVM means Low Level Virtual Machine, which is framework system of the architecture compiler, written in C++. The LLVM core library provides compiler related support and can be used as a background for multiple language compilers. And it can optimize the compilation time, link optimization, online compilation optimization and code generation of the programming language.

We downloaded the llvm 3.8 from the website and built the environment. Then we write a script to generate the LLVM IR. In the script, firstly, we striped the OpenCL file from file name, and then we copied the original file to another one, named llvm_tmp_cl, which was included in a C file. After that, we wrote three commands to obtain the llvm IR, which are as follows Fig.3.

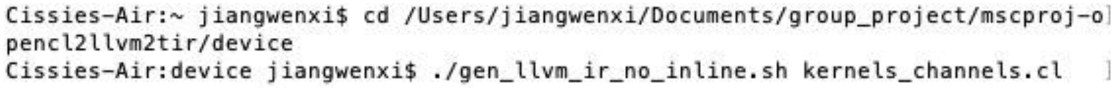


```
clang -O0 -S -Wunknown-attributes -emit-llvm -c  llvm_tmp.c -o llvm_tmp.unopt.ll
opt -mem2reg -S llvm_tmp.unopt.ll -o llvm_tmp.opt1.ll
clang  -O1 -S -emit-llvm llvm_tmp.opt1.ll -o $1.ll
```

Fig.3. LLVM+Clang+opt commands

The first command means to generate LLVM IR, but it is too redundant, including a lot of codes that we do not need, such as labels. Hence, we use the second command to optimize the original LLVM file to generate a simpler and more useful version where mem2reg pass converts non-SSA form of LLVM IR into SSA form, raising loads and stores to stack-allocated values to SSA values. Finally, we saved this to another file to guarantee the security of the file.

Then we executed the script in the terminal to generate the llvm IR. Fig.4.



```
Cissies-Air:~ jiangwenxi$ cd /Users/jiangwenxi/Documents/group_project/mscproj-o
pencl2llvm2tir/device
Cissies-Air:device jiangwenxi$ ./gen_llvm_ir_no_inline.sh kernels_channels.cl
```

Fig.4. The commands to execute the script

4.3 Transform the LLVM IR into *TyTra-IR*

After the generation of the LLVM IR, we transformed the LLVM IR into *TyTra-IR*, via a custom python script. Python is easy to learn and use with high speed and portability, hence, we choose the python to parse the LLVM IR into *TyTra-IR*.

After analyzing the format and the function meaning of each module of the input file, we establish a source to source complier system based on python 3.0. The input of the system is the generated LLVM IR file and the output is the *TyTra-IR* format file. We establish the system based on a simple given case, which only contain three kernels named as kernel input, kernel compute and kernel output. However, to achieve the complex task in real life, there would be more kernels and the name of each kernel could change as well. Thus, taking the above factors into consideration, while the programming the process, we have made our code as generic as it can be. Here, we will introduce the main idea behind the source to source system based on the given simple case. To be specifically, the framework of the source to source system is working based on the following flowchart shown in Fig.5. The evaluation of the effectiveness of the established system in the context of more complex situations will be presented in the evaluation part.

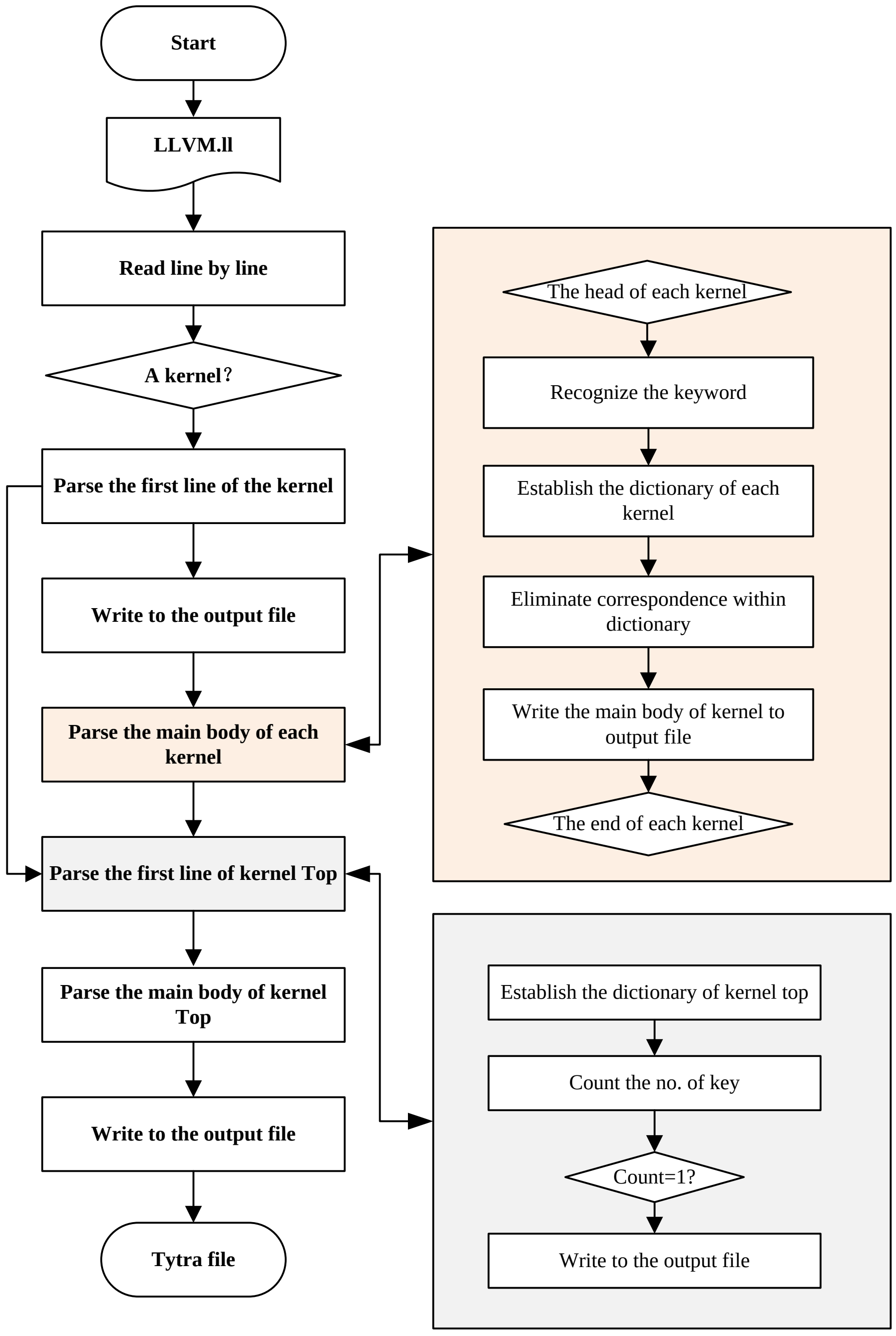

Start
LLVM.ll
Read line by line
A kernel?
Parse the first line of the kernel
Write to the output file
Parse the main body of each kernel
Parse the first line of kernel Top
Parse the main body of kernel Top
Write to the output file
Tytra file
The head of each kernel
Recognize the keyword
Establish the dictionary of each kernel
Eliminate correspondence within dictionary
Write the main body of kernel to output file
The end of each kernel
Establish the dictionary of kernel top
Count the no. of key
Count=1?
Write to the output file

Fig.5. The flowchart of the source to source compiler

The parser worked following steps of the left part of the chart. On the one hand, considering the different meaning and function between the first line and the main body part of each type of kernels, including kernel input, kernel compute(s) and kernel output, we have parsed each part separately and write sequentially to the output file. In particular, when parsing the main body of the above kernels, we have used the dictionary as an assistant to find the main body of each kernel. The main idea of the parse processing of the main body is specified in the right corner of the flowchart. On the other hand, because the relationship between the first lines of each kernel and the kernel top, which is supposed to be present in the output tytra file, we have used the first lines of each kernels to establish the kernel part and here we name it as words_for_top to elaborate. Since the first line of the kernel top is mainly functioned as the interface between global memory and FPGA, which means that the keywords in the first line of kernel top will appear only once. The main body of the kernel top is functioned as calling each kernel’ s function. Thus, there also are two steps in terms of the kernel top parsing. Firstly, we establish a dictionary based on the words_for_top and by counting the number of the key of the dictionary, we finalize the parsing of the first line of kernel top. Secondly, we parse the words_for_top lines to establish the main body part of the kernel top. After all the parsing process, an output file in the tytra format will get. The more detailed information about the parsing process can be found in the Appendix code file.

4.4 Generating the FPGA solution

Finally, we generate the correct TIRL file from a giving OpenCL kernel file that uses channels. The TIRL file can be passed to the *TyTra* back end compiler called *TyBEC*, which is provided by our professor Nabi. The *TyBEC* compiler automatically generates dataflow architecture from the TIRL file. The *TyBEC* also generates FPGA solution in the form of Verilog hardware description language (HDL) and OpenCL code (Nabi, 2018).

Chapter 5 Evaluation

5.1 Results of the Basic Case

Firstly, we had OpenCL file. The three main kernels are defined as follows:

Kernel input:

```
__kernelvoid kernelInput ( int aIn0
                         , int aIn1
                         , write_only pipe int     ch00
                         , write_only pipe int     ch01
                         ) {
int data0, data1;
//read from global memory
    data0 = aIn0;
    data1 = aIn1;
//write to channel
    write_pipe(ch00, &data0);
    write_pipe(ch01, &data1);
}
```

kernel compute:

```
__kernelvoid kernelCompute (
read_only  pipe int ch00
                           ,read_only  pipe int ch01
                           ,write_only pipe int ch1
                           ) {

//locals
int dataIn0, dataIn1, dataOut;
int i;
//read from channels
    read_pipe(ch00, &dataIn0);
    read_pipe(ch01, &dataIn1);

//the computation
    dataOut = dataIn0 + dataIn1;

//write to channel
    write_pipe(ch1, &dataOut);
}
```

kernel output:

```
__kernelvoid kernelOutput( int aOut
                         , read_only pipe int ch1
                         ) {
int data;
//read from channel
    read_pipe(ch1, &data);
//write to global mem
    aOut = data;
}
```

Secondly, we used the script convert input OpenCL code in to LLVM IR.

Thirdly, we parsed the LLVM IR via python script and obtained the *TyTra-IR*. The following is the results of our parser worked on the simple example case with three kernels named as: kernel input, kernel compute and kernel output. The generated results that produced by our parser is same as the expected *TyTra-IR* results.

To be specific, the generated kernel input of our compiler is:

```
define void @kernelInput(i32 %aIn0, i32 %aIn1, i32 %ch00, i32 %ch01) pipe  {
 i32 %ch00 = load i32 %aIn0
 i32 %ch01 = load i32 %aIn1
 ret void
}
```

In the kernel input, the channel named as ch00 loaded the value named as aIn0 from the memory. In the same way, another channel named as ch01 loaded the value named as aIn1. And the generated kernel compute is:

```
define void @kernelCompute(i32 %ch00, i32 %ch01, i32 %ch1) pipe  {
i32 %ch1 =add i32 %ch00, %ch01,
 ret void
}
```

The kernel compute conducts a simple function: add the value of ch00 which is aIn0 and the value of ch01 which is aIn1 together, and the result is stored in the ch1. And the generated kernel output is:

```
define void @kernelOutput(i32  %aOut, i32 %ch1) pipe  {
 i32 %aOut = load i32 %ch1
 ret void
}
```

The kernel output loaded the computed result of the kernel compute, ch1 as it mentioned in the kernel compute, to aOut, which is connected to the memory. And the generated kernel top is:

```
define void @kernelTop (i32 %aIn0, i32 %aIn1, i32 %aOut,)pipe {
call @kernelInput(i32%aIn0,i32%aIn1,i32%ch00,i32%ch01)
call @kernelCompute(i32%ch00,i32%ch01,i32%ch1)
call @kernelOutput(i32%aOut,i32%ch1)
ret void
}
```

The kernel top is generated to call all the functions to conduct the computations.

Finally, we had the FPGA solution. The following dataflow graphs are generated by the "*TyTra* back-end compiler" (*TyBEC*) and represent the FPGA circuit that *TyBEC* generates for the given TIR file. By analyzing the logic of these dataflow graphs, it can be reached a conclusion that out project is generating correct TIR code based on the basic case.

To the specific, the data flow of kernel Input is as shown in Fig.6:

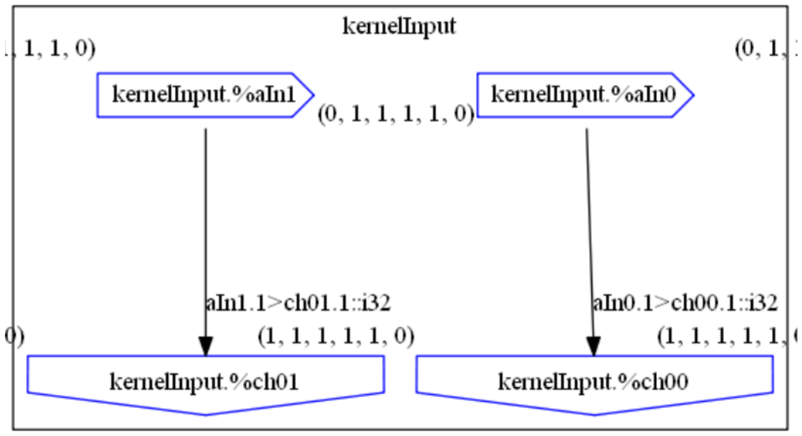


Fig.6. Data flow of kernel input of basic case

The data flow of kernel compute is as shown in Fig.7:

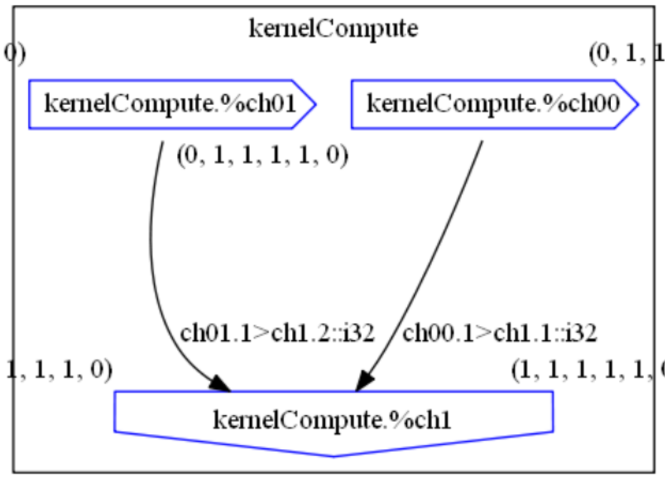


Fig.7. Data flow of kernel compute of basic case

The dataflow of kernel output is as shown in Fig.8:

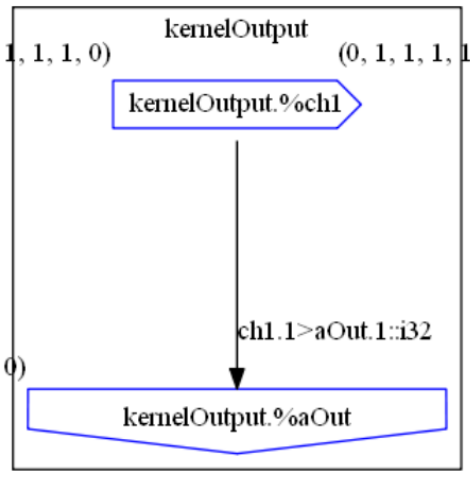


Fig.8. Data flow of kernel output of basic case

And the dataflow of kernel Top is as shown in Fig.9:

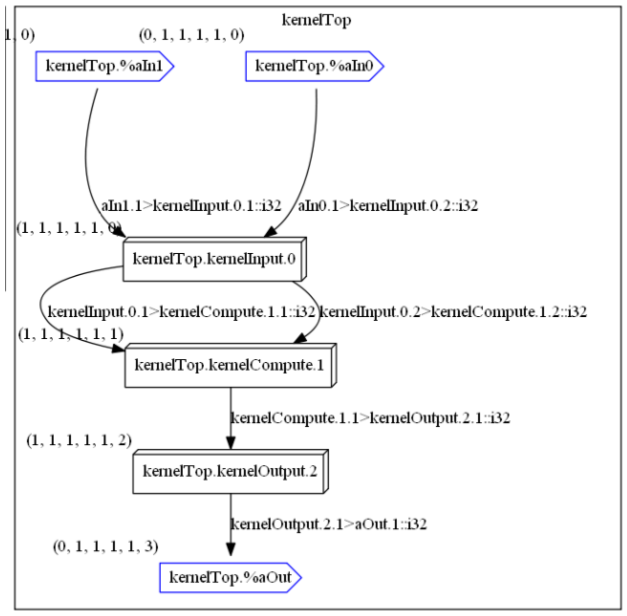

kernelTop
1, 0)
(0, 1, 1, 1, 1, 0)
kernelTop.%aIn1
kernelTop.%aIn0
aIn1.1>kernelInput.0.1::i32
aIn0.1>kernelInput.0.2::i32
(1, 1, 1, 1, 1, 0)
kernelTop.kernelInput.0
kernelInput.0.1>kernelCompute.1.1::i32
kernelInput.0.2>kernelCompute.1.2::i32
(1, 1, 1, 1, 1, 1)
kernelTop.kernelCompute.1
kernelCompute.1.1>kernelOutput.2.1::i32
(1, 1, 1, 1, 1, 2)
kernelTop.kernelOutput.2
kernelOutput.2.1>aOut.1::i32
(0, 1, 1, 1, 1, 3)
kernelTop.%aOut

Fig.9. Data flow of kernel top of basic case

5.2 Further Testing

The results of the basic case showed that our project can work well, thus, we exploited tree debug cases to test the project. The generated results correctly represent the functions we defined. The first case is to add another function in kernel compute, because in the real-life context, the computing task would more complicated, not only a simple add function but also other more complicated ones like another add/multiply functions or their combination. The second case is to add additional input to first kernel and add another compute instruction. The third one is to evaluate the effectiveness of our parse with the situation where multiple kernels used to compute. Here, we added another kernel to the pipeline. The performances of the above debug case studies are presented in the following:

The first debug case adds another compute instruction (like another add/multiply in kernel Compute). The generated tytra IR result is:

```
define  void  @kernelInput(i32  %aIn0,  i32  %aIn1,  i32  %ch00,  i32
%ch01) pipe  {
 i32 %ch00 = load i32 %aIn0
 i32 %ch01 = load i32 %aIn1
 ret void
}
define void @kernelCompute(i32 %ch00, i32 %ch01, i32 %ch1) pipe  {
i32 %5 =mul i32 %ch00, %ch01,
i32 %ch1 =mul i32 %5, %ch01,
 ret void
}
define void @kernelOutput(i32  %aOut, i32 %ch1) pipe  {
 i32 %aOut = load i32 %ch1
 ret void
}

define void @kernelTop (i32 %aIn0, i32 %aIn1, i32 %aOut,) pipe {
call @kernelInput(i32%aIn0,i32%aIn1,i32%ch00,i32%ch01)
call @kernelCompute(i32%ch00,i32%ch01,i32%ch1)
call @kernelOutput(i32%aOut,i32%ch1)
ret void
}
```

And the produced dataflow by the "*TyTra* back-end compiler" is as follows:

Input kernel is as shown in Fig.10:

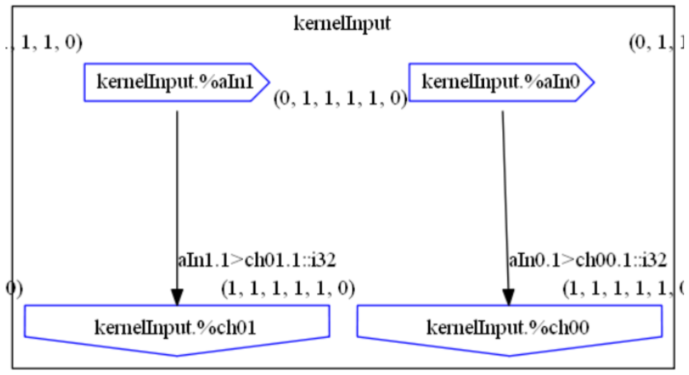


Fig.10. Data flow of kernel input of the first debug case

Compute kernel is as shown in Fig.11:

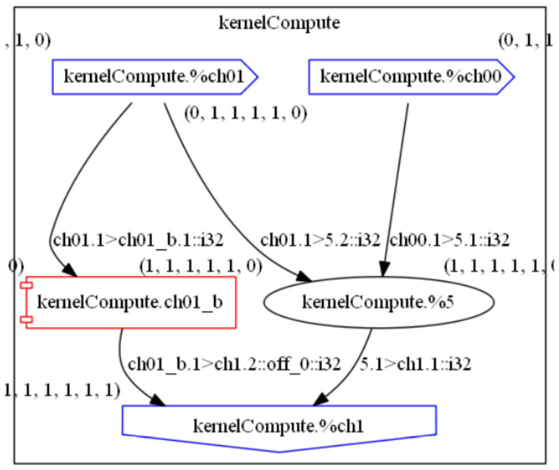


Fig.11. Data flow of kernel compute of the first debug case

Output kernel is as shown in Fig.12:

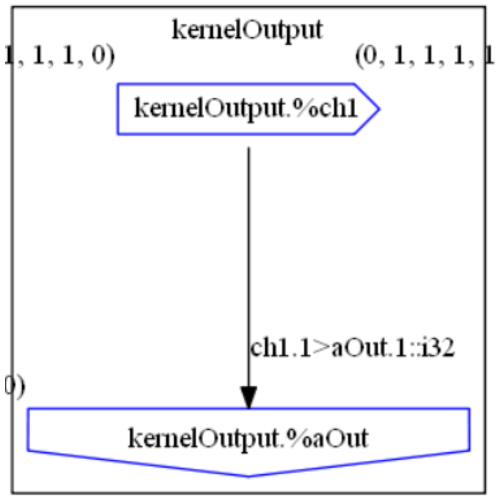


Fig.12. Data flow of kernel output of the first debug case

And kernel Top is as shown in Fig.13:

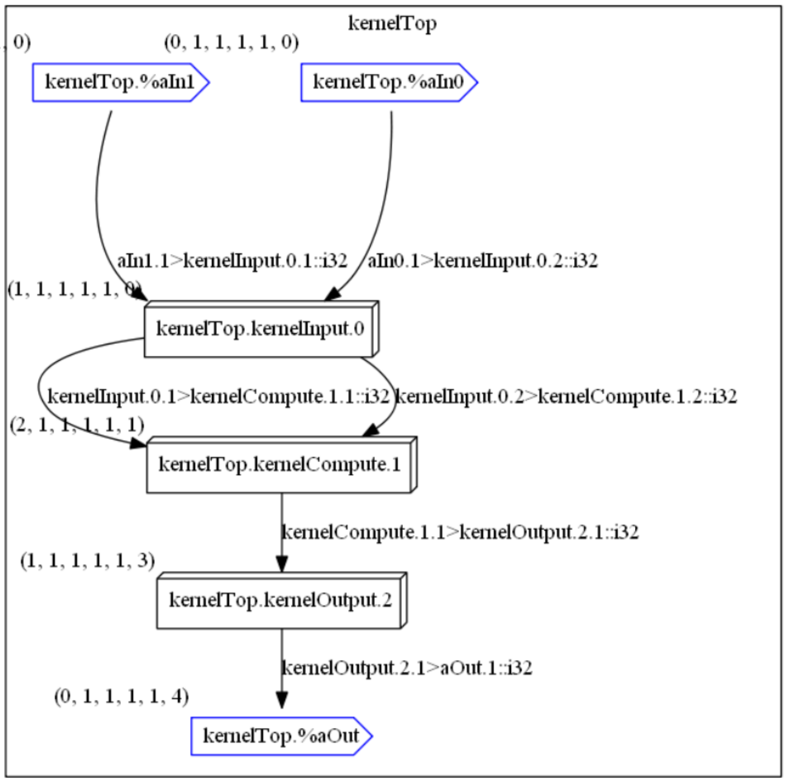

kernelTop
, 0)
(0, 1, 1, 1, 1, 0)
kernelTop.%aIn1
kernelTop.%aIn0
aIn1.1>kernelInput.0.1::i32
aIn0.1>kernelInput.0.2::i32
(1, 1, 1, 1, 1, 0)
kernelTop.kernelInput.0
kernelInput.0.1>kernelCompute.1.1::i32
kernelInput.0.2>kernelCompute.1.2::i32
(2, 1, 1, 1, 1, 1)
kernelTop.kernelCompute.1
kernelCompute.1.1>kernelOutput.2.1::i32
(1, 1, 1, 1, 1, 3)
kernelTop.kernelOutput.2
kernelOutput.2.1>aOut.1::i32
(0, 1, 1, 1, 1, 4)
kernelTop.%aOut

Fig.13. Data flow of kernel top of the first debug case

The second case is to add additional input to first kernel and add another compute instruction. The generated *TyTra-IR* result is:

```
define  void  @kernelInput(i32  %aIn0,  i32  %aIn1,  i32  %aIn2,  i32
%ch00, i32 %ch01, i32 %ch02) pipe  {
 i32 %ch00 = load i32 %aIn0
 i32 %ch01 = load i32 %aIn1
 i32 %ch02 = load i32 %aIn2
 ret void
}
define  void  @kernelCompute(i32  %ch00,  i32  %ch01,  i32  %ch02,  i32
%ch1) pipe  {
i32 %6 =add i32 %ch00, %ch01,
i32 %ch1 =add i32 %6, %ch02,
 ret void
}
define void @kernelOutput(i32  %aOut, i32 %ch1) pipe  {
 i32 %aOut = load i32 %ch1
 ret void
}
define  void  @kernelTop  (i32  %aIn0,  i32  %aIn1,  i32  %aIn2,  i32
%aOut,  )pipe {
call
@kernelInput(i32%aIn0,i32%aIn1,i32%aIn2,i32%ch00,i32%ch01,i32%ch02
)
call @kernelCompute(i32%ch00,i32%ch01,i32%ch02,i32%ch1)
call @kernelOutput(i32%aOut,i32%ch1)
ret void
}
```
And the produced dataflow by the "*TyTra* back-end compiler" is as follows:

Input kernel is as shown in Fig.14:

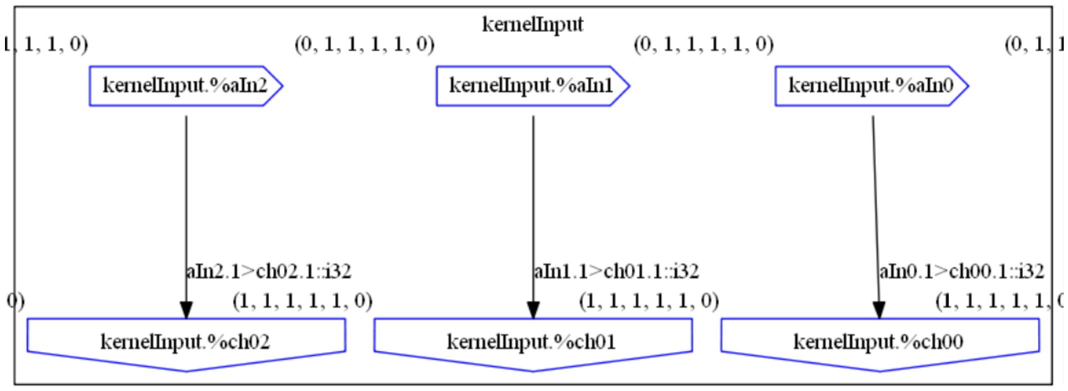


Fig.14. Data flow of kernel input of the second debug case

Kernel compute is as shown in Fig.15:

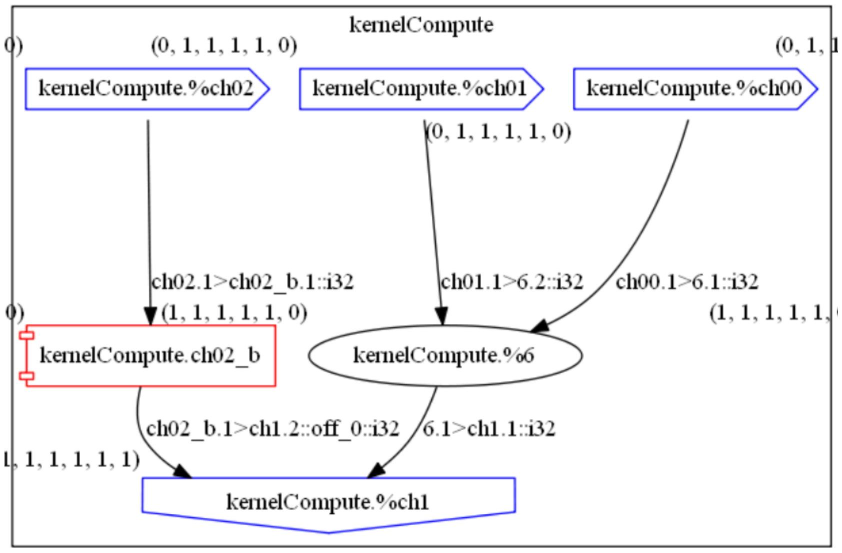


Fig.15. Data flow of kernel compute of the second debug case

Kernel output is as shown in Fig.16:

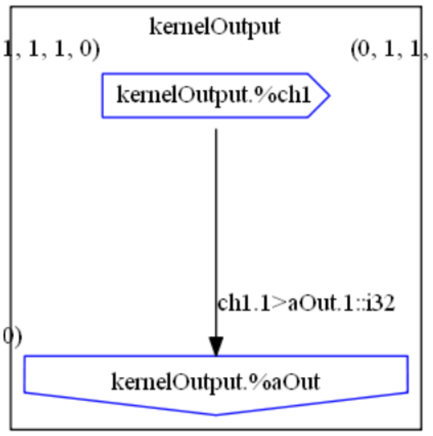
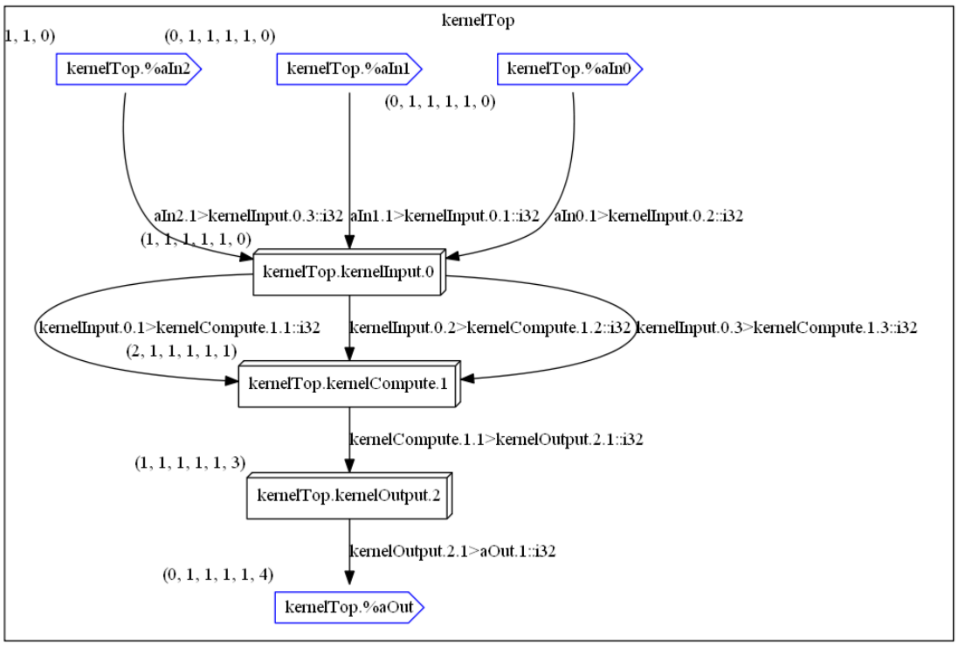


Fig.16. Data flow of kernel compute of the second debug case

And kernel Top is as shown in Fig.17:



kernelTop
1, 1, 0)
(0, 1, 1, 1, 1, 0)
kernelTop.%aIn2
kernelTop.%aIn1
kernelTop.%aIn0
(0, 1, 1, 1, 1, 0)
aIn2.1>kernelInput.0.3::i32
aIn1.1>kernelInput.0.1::i32
aIn0.1>kernelInput.0.2::i32
(1, 1, 1, 1, 1, 0)
kernelTop.kernelInput.0
kernelInput.0.1>kernelCompute.1.1::i32
kernelInput.0.2>kernelCompute.1.2::i32
kernelInput.0.3>kernelCompute.1.3::i32
(2, 1, 1, 1, 1, 1)
kernelTop.kernelCompute.1
kernelCompute.1.1>kernelOutput.2.1::i32
(1, 1, 1, 1, 1, 3)
kernelTop.kernelOutput.2
kernelOutput.2.1>aOut.1::i32
(0, 1, 1, 1, 1, 4)
kernelTop.%aOut

Fig.17. Data flow of kernel top of the second debug case

The third one is to evaluate the effectiveness of our parse with the situation where multiple kernels used to compute. Here, we add another kernel to the pipeline. The generated *TyTra-IR* result is:

```
define void @kernelInput(i32 %aIn0, i32 %aIn1, i32 %ch00a, i32
%ch01a) pipe {
 i32 %ch00a = load i32 %aIn0
 i32 %ch01a = load i32 %aIn1
 ret void
}
define void @kernelComputeA(i32 %ch00a, i32 %ch01a, i32 %ch1a)
pipe {
i32 %ch1a =mul i32 %ch00a, %ch01a,
 ret void
}
define void @kernelComputeB(i32 %ch1a, i32 %ch1b) pipe  {
i32 %ch1b =add i32 %ch1a, %ch1a,
 ret void
}
define void @kernelOutput(i32  %aOut, i32 %ch1b) pipe  {
 i32 %aOut = load i32 %ch1b
 ret void
}

define void @kernelTop (i32 %aIn0, i32 %aIn1, i32 %aOut,  )pipe {
call @kernelInput(i32%aIn0,i32%aIn1,i32%ch00a,i32%ch01a)
call @kernelComputeA(i32%ch00a,i32%ch01a,i32%ch1a)
call @kernelComputeB(i32%ch1a,i32%ch1b)
call @kernelOutput(i32%aOut,i32%ch1b)
ret void
}
```

And the produced dataflow by the "*TyTra* back-end compiler" is as follows:

Kernel input is as shown in Fig.18:

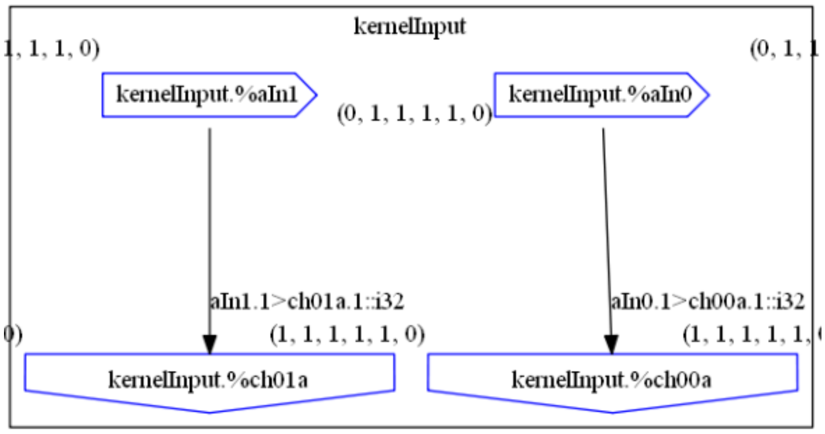
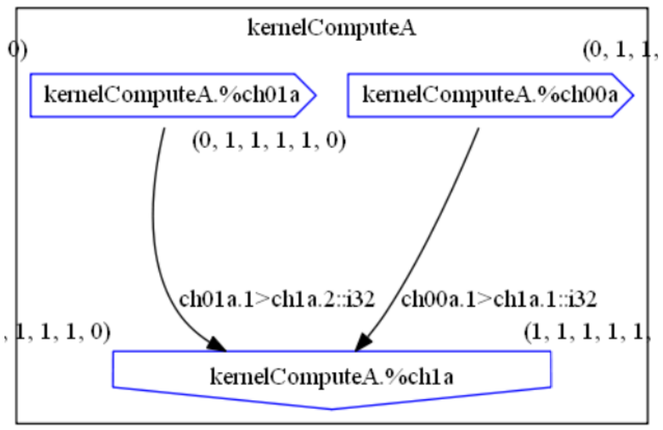


Fig.18. Data flow of kernel input of the third debug case

Kernel Compute is as shown in Fig.19:



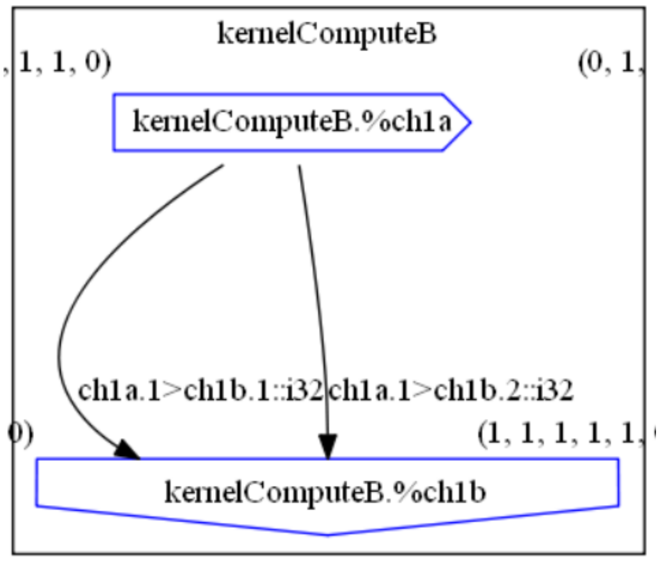


Fig.19. Data flow of kernel input of the third debug case

Kernel output is as shown in Fig.20:

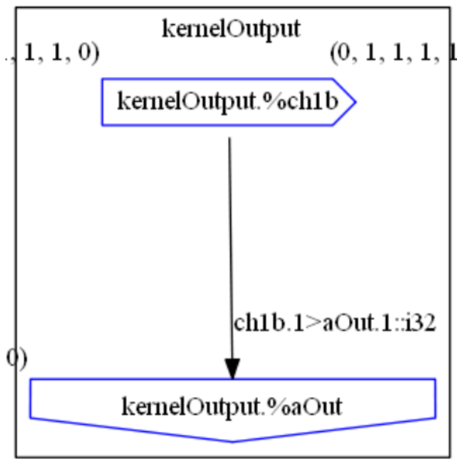


Fig.20. Data flow of kernel output of the third debug case

And Kernel Top is as shown in Fig.21:

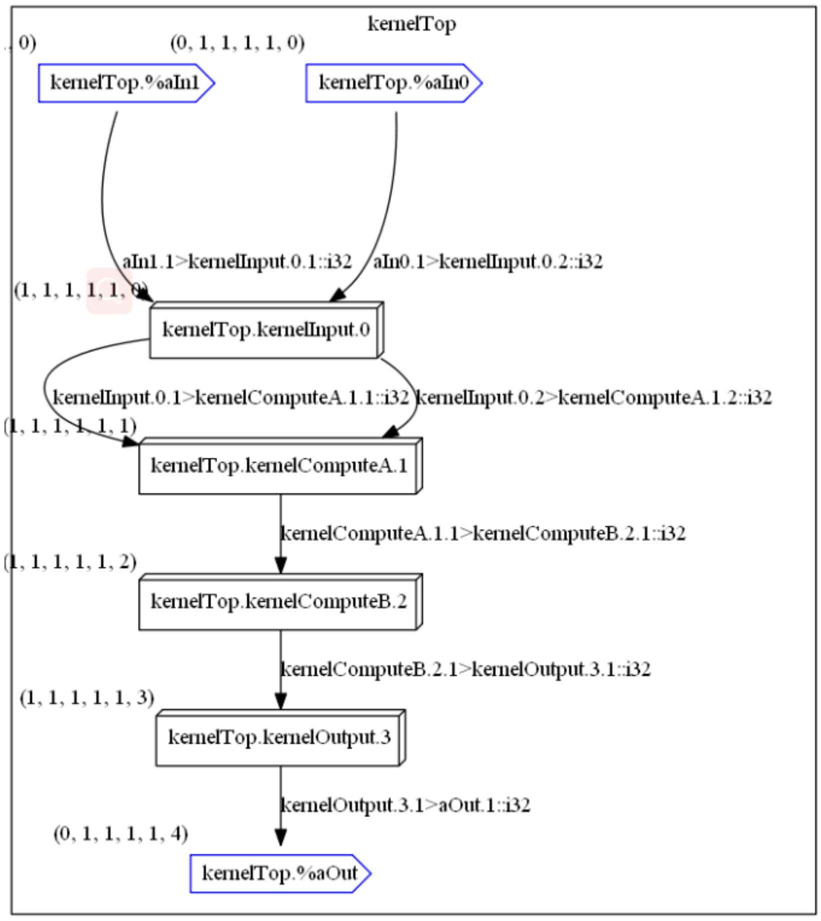

kernelTop
. 0)
(0, 1, 1, 1, 1, 0)
kernelTop.%aIn1
kernelTop.%aIn0
aIn1.1>kernelInput.0.1::i32
aIn0.1>kernelInput.0.2::i32
(1, 1, 1, 1, 1, 0)
kernelTop.kernelInput.0
kernelInput.0.1>kernelComputeA.1.1::i32
kernelInput.0.2>kernelComputeA.1.2::i32
(1, 1, 1, 1, 1, 1)
kernelTop.kernelComputeA.1
kernelComputeA.1.1>kernelComputeB.2.1::i32
(1, 1, 1, 1, 1, 2)
kernelTop.kernelComputeB.2
kernelComputeB.2.1>kernelOutput.3.1::i32
(1, 1, 1, 1, 1, 3)
kernelTop.kernelOutput.3
kernelOutput.3.1>aOut.1::i32
(0, 1, 1, 1, 1, 4)
kernelTop.%aOut

Fig.21. Data flow of kernel top of the third debug case

By analyzing the logic of these dataflow graphs, it can be reached a conclusion that out project is generating correct TIR code based on the universal case studies

Chapter 6 Conclusion

In this paper, we had introduced the aims of our project, which is to develop a series of scripts that allow transform pipe-based OpenCL kernels to *TyTra-IR* programs. We also gave some background survey on the main sections which we are interested in and can help our research, such as *TyTra* project and LLVM. Then we presented the method that we used on details. Lastly, we displayed the achievements obtained by our project and more studies.

In this project, we have researched how to design a *TyTra* project flow to make it easier to deploy legacy scientific code onto heterogeneous systems, in particular on FPGAs, and how to use LLVM to transform different languages. Moreover, we have successfully achieved a python source-source translator, which can be widely used to transform LLVM code to tirl file that can be used on FPGA directly.

In the future, we hope we can enhance our translator to deal with more complicated and practical problems, and we hope our programming flow can be applied to more general problems and do more study beyond our project, for example, optimizing the performance of FPGAs.

Chapter 7 Contributions

Xiao Wang and Songpei Xu wrote the OpenCL code and wrote a python source-source- translator to translate LLVM IR into the *TyTra-IR*. And wrote and modified the dissertation.

Jing Pan transformed C code in to the LLVM IR. And wrote and modified the dissertation.

Sijia Niu collected related papers for us.

References

[1] Czajkowski, T. S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., ... & Singh, D. P. (2012, August). From OpenCL to high-performance hardware on FPGAs. In Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on (pp. 531-534). IEEE

[2] https://clang.llvm.org/ last available on 11/21/2018.

[3] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (p. 75). IEEE Computer Society.

[4] Lattner, C. (2008, May). LLVM and Clang: Next generation compiler technology. In The BSD conference (pp. 1-2).

[5] Nabi. S. w. (2018). *The meeting fragment.*

[6] Nabi, S. W., & Vanderbauwhede, W. (2017). FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis. Journal of Parallel and Distributed Computing.

[7] Nabi, S. W., & Vanderbauwhede, W. (2017). Type-driven automated program transformations and cost modelling for optimising streaming programs on FPGAs. International Journal of Parallel Programming.

[8] Shagrithaya, K., Kepa, K., & Athanas, P. (2013, June). Enabling development of OpenCL applications on FPGA platforms. In 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (pp. 26-30). IEEE.

[9] Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3), 66-7