Julia meets the FPGA : Higher-level synthesis methodology forheterogeneous hardware and software architectures

Gaëtan Lounes, Robin Gerzaguet, Matthieu Gautier

Julia meets the FPGA

Higher-level synthesis methodology for heterogeneous hardware and software architectures

Gaëtan LOUNES*, Robin GERZAGUET*, Matthieu GAUTIER*
*Univ Rennes, CNRS, IRISA, firstname.name@irisa.fr

Introduction

Julia language has acknowledged great results in the heterogeneous compilation place due to the flexibility of the compilation toolchain and thanks to extension of this compiler. For instance GPU and IPU can be targeted smoothly. FPGA are another class of interesting target which has been little-explored [1], they aim to operate in energy efficient and high throughput applications. **We propose Judias: A Julia MLIR frontend for HLS**.

Glossary

AST: Abstract Syntax Tree

Brutus: Brutus is a research project to bring MLIR to Julia (https://github.com/JuliaLabs/brutus)

Clang: LLVM backend C-like compiler (https://clang.llvm.org/)

DSE: Design Space Exploration

DSL: Design Specific Language

FPGA: Field Programmable Gate Array

HDL: Hardware Description Language

HLS: High Level Synthesis

IR: Intermediate Representation

LLVM: Low Level Virtual Machine, State of the art compiler backend (https://llvm.org/) [2]

MLIR: Multi Layer IR (https://mlir.llvm.org/) [3]

SSA: Static Single Assignment

Julia compiler heterogeneous hardware usages

To support new target, the GPUCompiler.jl[1] package is used. It reuses the plain Julia compiler up to the translate step. Then in the backend, each target defines a new LLVM pipeline for their specific needs in order to generate specialized LLVMIR.

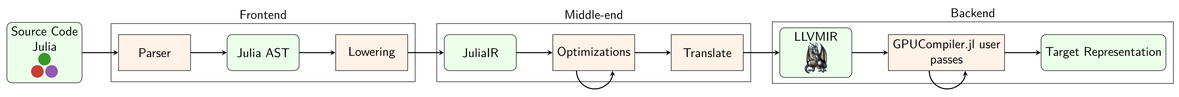

Figure 1: Simplified Julia toolchain using GPUCompiler.jl

HLS

FPGA are configured using HDL language such as VHDL or Verilog. But this form is not ideal because it requires solid hardware knowledge and algorithm development is unwieldy. HLS have been introduced to overcome these limitations. State of the art, HLS tools, such as Vitis from AMD, uses a DSL based on C (using a custom Clang) which is then transformed to a LLVMIR containing various annotations to model hardware semantics. The closed-source nature of this IR usage and semantic make its integration with new frontend cumbersome.

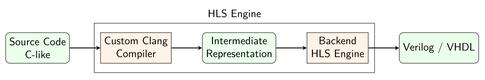

Figure 2: Simplified HLS toolchain of Vitis

A new compiler framework: MLIR

An alternative method used by many HLS users doesn't use the DSL or LLVM IR directly; they introduced a new toolchain using MLIR [3]. The idea behinds MLIR is to generalize the concept of IR and ways to used and mutated it. They introduced new IR (called dialects in MLIR terminology) which are analyzed and changed to fit an IR which is then sent to HLS tools.

MLIR key principles

- Expressively: Dialect / Attributes / Operator / Type / Region
- Reusability: Traits / Pass / Compiler
- Transformable: Progressive Lowering / Rewrite Patterns

Brutus: a MLIR Julia backend

"Brutus is a research project that uses MLIR to implement code-generation and optimisations for Julia."

Brutus backend has some singularities:

- Julia IR is the input of Brutus. Contrary to GPUCompiler.jl which process LLVMIR.
- Aim to improve backend compilations pass.

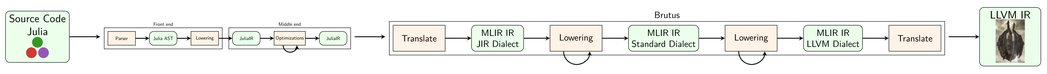

Figure 3: A simple Brutus toolchain

Judias

In a similar manner as Brutus prototype tool, we propose using Julia IR as MLIR frontend.

The key difference lies in the usage of MLIR, it's not for improving backend compilation performance but to be used as a frontend of a HLS user. In this case: ScaleHLS [4]. It is a HLS framework which uses MLIR extensively to represent and transform HLS designs at different levels of abstraction. Ultimately, the final representation created by the framework is transformed in order to generate code usable by Vitis.

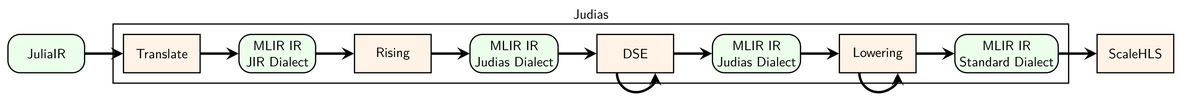

Figure 4: Judias toolchain

Julia compiler example

Listing 1 shows a simple Julia function containing arithmetic, local definition and a for loop. This program goes through the Julia toolchain and Listing 2 is the resulting form just before translate to LLVM form.

```
1 function main(N::Int64)
2   acc2 = 1
3   acc = 1
4   for i in 1:N
5     acc += i
6     acc2 += 2*i
7   end
8   return acc + acc2
9 end
```

Listing 1: a Julia program

```
1  1 ─ %1    =  (1:_2)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1), Int64])
2  │   %2  = Base.iterate(%1)::Union{Nothing, Tuple{Int64, Int64}}
3  │   %3  = (%2 === nothing)::Bool
4  │   %4  = Base.not_int(%3)::Bool
5  └──       goto #4 if not %4
6  2 ─ %6  = φ (#1 => %2, #3 => %14)::Tuple{Int64, Int64}
7  │   %7  = φ (#1 => 1, #3 => %11)::Int64
8  │   %8  = φ (#1 => 1, #3 => %13)::Int64
9  │   %9  = Core.getfield(%6, 1)::Int64
10 │   %10 = Core.getfield(%6, 2)::Int64
11 6 │   %11 = Base.add_int(%7, %9)::Int64
12 7 │   %12 = Base.mul_int(2, %9)::Int64
13 │   %13 = Base.add_int(%8, %12)::Int64
14 8 │   %14 = Base.iterate(%1, %10)::Union{Nothing, Tuple{Int64, Int64}}
15 │   %15 = (%14 === nothing)::Bool
16 │   %16 = Base.not_int(%15)::Bool
17 └──       goto #4 if not %16
18 3 ─       goto #2
19 9 4 ─ %19 = φ (#2 => %11, #1 => 1)::Int64
20 │   %20 = φ (#2 => %13, #1 => 1)::Int64
21 │   %21 = Base.add_int(%19, %20)::Int64
22 └──       return %21
```

Listing 2: IR generated

Julia IR remarks

- Julia IR contains standard structure (mainly shared with LLVM):
  - SSA
  - phinode
  - basic blocks
  - strongly typed
- Explicit loop information is erased
- Non-standard compilation toolchain: Inliner is changed to keep track of some functions calls.

Judias usage

The powerful framework MLIR enables the write of a dialect for Julia. Julia IR is first changed to fit with this dialect:

- phinodes are replaced by block with arguments
- Julia type and operation behaviors are mapped to the MLIR dialect
- instructions are translated to MLIR operations

Listing 3 shows the result of passes which was executed to transform the dialect IR, for instance to retrieve the spatial loop informations and to canonicalize the IR.

```
1  module {
2    func.func @main(%arg0: !JIR.JC<Int64>) -> !JIR.JC<Int64> {
3     JIR.goto[^bb1] : ()
4     ^bb1:  // pred: ^bb0
5      %0:2 = "JIR.loop"() ({
6       %2 = JIR.ConstantOp {value = #JIR.s<1>} : (!JIR.JC<Int64>)
7       JIR.ret(%2,%2) : (!JIR.JC<Int64>,!JIR.JC<Int64>)},
8      {
9       %2 = JIR.ConstantOp {value = #JIR.s<1>} : (!JIR.JC<Int64>)
10      %3 = "JIR.colon"(%2, %arg0) : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> !JIR.JC<Base.UnitRange{Int64}>
11      JIR.ret(%3) : (!JIR.JC<Base.UnitRange{Int64}>)},
12     {
13     ^bb0(%arg1: !JIR.JC<Int64>, %arg2: !JIR.JC<Int64>, %arg3: !JIR.JC<Int64>):
14      %2 = JIR.IntrinsicOp(%arg1, %arg3) {f = #JIR.s<#<add_int>>} : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> !JIR.JC<Int64>
15      %3 = JIR.ConstantOp {value = #JIR.s<2>} : (!JIR.JC<Int64>)
16      %4 = JIR.IntrinsicOp(%3, %arg3) {f = #JIR.s<#<mul_int>>} : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> !JIR.JC<Int64>
17      %5 = JIR.IntrinsicOp(%arg2, %4) {f = #JIR.s<#<#2 add_int>>} : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> !JIR.JC<Int64>
18      "JIR.yield"(%2, %5) : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> ()
19     }) : () -> (!JIR.JC<Int64>, !JIR.JC<Int64>)
20   %1 = JIR.IntrinsicOp(%0#0, %0#1) {f = #JIR.s<#<add_int>>} : (!JIR.JC<Int64>, !JIR.JC<Int64>) -> !JIR.JC<Int64>
21   JIR.ret(%1) : (!JIR.JC<Int64>)
22   }
23 }
```

Listing 3: MLIR using JIR custom dialect

Feature	Status
Julia IR parser	✅
Julia type system in MLIR	🚧
Julia intrinsic / builtins	🚧
Raising loop pass	✅
Lowering Pass to standard MLIR	🚧
Arrays	🚧
User function call	🏗️
Precise effects semantic	🏗️
DSE	🏗️

Judias remarks

- Loop spatial information is retrieved during rising pass and is explicitely defined in the dialect
- The JIR dialect contains runtime information from Julia interpreter such as types or function call

To standard dialect and Hardware

Listing 3 can be lowered to standard dialect (Affine, Arith, Function, Index) to obtain Listing 4, then in this form it can be provided to the HLS framework. ScaleHLS and then Vitis HLS generated a HDL representation using Table 1 ressources and which can be represented as Figure 5.

```
1  module {
2    func.func @main(%arg0: i64) -> i64 {
3      %0 = arith.constant 1 : i64
4      %1 = arith.addi %arg0, %0 : i64
5      %2 = arith.index_cast %1 : i64 to index
6      %3:2 = affine.for %i = 1 to %2 iter_args(%arg1 = %1, %arg2 = %1) -> (i64, i64) {
7        %4 = arith.index_cast %i : index to i64
8        %5 = arith.addi %arg1, %4 : i64
9        %6 = arith.constant 2 : i64
10       %7 = arith.muli %6, %4 : i64
11       %8 = arith.addi %arg2, %7  : i64
12       affine.yield %5, %8 : i64, i64}
13     %9 = arith.addi %3#0, %3#1 : i64
14     return %9 : i64}
15 }
```

Listing 4: MLIR using standard dialect

Ressource	Quantity
LUT	228
FF	69
DSP	4
BRAM	2
URAM	0

Table 1: RTL ressource usage

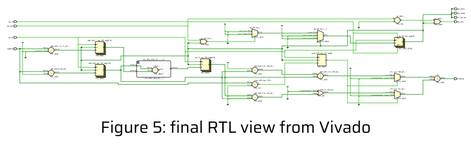

Figure 5: final RTL view from Vivado

Conclusion

Julia can be used with FPGA through HLS. Notably, Julia IR singularities (typed IR and compactness) are bringed to MLIR using the powerness of Julia compilation.

One major extend is to bring a DSE into the playground: using Julia interpreter capacities and MLIR lively semantics to achieves a rich and fast design space analysis. One idea is to exploit Graph Neural Network on MLIR representation.

Bibliography

[1] B. Biggs, I. McInerney, E. C. Kerrigan, and G. A. Constantinides, "High-Level Synthesis Using the Julia Language," no. arXiv:2201.11522. arXiv, Feb. 2022.

[2] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *Languages and Compilers for High Performance Computing*, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 15–16. doi: 10.1007/11532378_2.

[3] C. Lattner *et al.*, "MLIR: A Compiler Infrastructure for the End of Moore's Law," no. arXiv:2002.11054. arXiv, Feb. 2020.

[4] H. Ye *et al.*, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," no. arXiv:2107.11673. arXiv, Dec. 2021.

This work is funded by ANR under the grant number ANR-22-CE25-0007-01 (RedInBlack Project)