CertSkel: a Verified Compiler for a Coq-embedded GPGPU DSL

Izumi Asakura Hidehiko Masuhara Tomoyuki Aotani
Tokyo Institute of Technology, Tokyo, Japan
asakura.i.aa@m.titech.ac.jp masuhara@acm.org aotani@is.titech.ac.jp

1. Introduction

GPUDSLs (domain specific languages for GPGPU) such as Accelerate [6], Ikra [7] and Copperhead [4] offer high-level data-parallel skeletons such as `map` and `reduce`, which are executed in parallel on GPUs. Those languages enable easier development of highly-optimized GPGPU programs by hiding complicated parallel memory access code behind compilers.

We propose a formally verified compiler for a GPUDSL, called *CertSkel*, that guarantees the correctness of GPGPU programs in the presence of advanced optimizations. Although there have been successful verified compilers (e.g., CompCert [5]), it is still challenging to develop a verified compiler for GPUDSL due to its parallel execution model and template-based code generation.

This paper gives an overview of CertSkel. We show the Coq-based compiler implementation (Section 2), our definition of the correctness of a template-based compiler (Section 3), and our approach to correctness proof with development of a tactic library (Section 4). We also discuss technical challenges left for the future work (Section 5).

2. Coq-Based Compiler Implementation

CertSkel is implemented in Coq, and generates a GPGPU program from a GPUDSL function. We use the theorem proving features in Coq for code generation, so that the generated code is correct with respect to the source function.

Figure 1 shows a source function and a compilation process in CertSkel. The Coq function `argmax` is CertSkel source code, which computes an index of the largest element in `xs` and the element itself. The expression (`seq` n m) at line 3 generates a list $[n, n+1, \ldots, n+m-1]$. A source function may use skeletons (including `map`, `reduce`, `zip`, and `seq`). We call the functions applied to skeletons *sequential functions*.

```
1  Definition argmax xs :=
2    reduce (fun (i,x) (j,y) ⇒ if y ≤ x then (i,x) else (j,y))
3      (zip (seq 0 (length xs)) xs)
4  Definition argmaxGPGPU: {p : GPGPU.prog | argmax ≈CG p}.
5  Proof.
6    reifyFunc.
7    eexists; apply compileOK.
8  Defined.
9  Definition res := generateGPGPUFile "./argmax.cu" argmaxGPGPU.
10 Separate Extract res.
```

Figure 1. Definition of `argmax` and GPGPU code generation

From a source function, CertSkel generates a GPGPU program by proving the following lemma (`argmaxGPGPU`): *there exists a GPGPU program that is equivalent to the source function* (line 4). The relation $f \approx_{CG} p$ means that a Coq function f and a GPGPU program p are *equivalent*, which we will explain in Section 3.

For manipulating the syntax tree of the source function, CertSkel reifies the source function to a typed syntax tree (called *TypedIR*) by using the `reifyFunc` tactic (line 6). A TypedIR function consists of a sequence of let-bindings, each of which has an application of a skeleton whose arguments are either sequential functions, scalar expressions, or variables holding lists. The last expression of the sequence is a variable. For example, `argmax` is reified to the following TypedIR function namely `argmaxR`:

```
fun xs ⇒ let a0 := seq 0 (length xs) in
  let a1 := zip a0 xs in
  let a2 := reduce (fun (i,x) (j,y)⇒ ...) a1 in a2
```

where ... is the syntax tree equivalent to the `argmax`'s one. The proof obligation after the line 6 is {`p : GPGPU.prog` | `argmaxR` $\approx_{IG}$ `p`} where $\approx_{IG}$ is the relation of equivalence between TypedIR functions and GPGPU programs.

We apply the `compileOK` lemma to obtain the GPGPU code (line 7). The statement of the lemma is `forall f : TypedIR, f` $\approx_{IG}$ `compileIR f`. The function `compileIR` compiles each skeleton application to a GPGPU kernel (a function executed by each GPU thread) and generates host code that allocates GPU memories and launches the kernels. By applying `compileOK` to the goal, `argmaxGPGPU` is proved.

Lines 9–10 just output the compiled code into a text file by using the Coq extraction mechanism.

3. Correctness

Our definition of correctness is based on GPUCSL, a concurrent separation logic for GPGPU kernels [1]. A judgement $\vdash \{P\}c\{Q\}$ in the system intuitively means that if the program c starts with a state satisfying the precondition P, then c does not cause any error during the execution and any final state satisfies the postcondition Q. Its soundness is formally proved in Coq.

The relation of equivalence $f \approx_{CG} p$ (and similarly $f_{IR} \approx_{IG} p$) is defined as follows.

$$\frac{\mathit{safe}(f, l)}{\vdash \begin{array}{c}\{\mathsf{array}(\mathtt{inp}, l)\}\\ p(\mathtt{inp})\\ \{\mathsf{array}(\mathtt{inp}, l) \star \mathsf{array}(\mathtt{ret}, f(l))\}\end{array}}$$

It means that for any input list l, if the application $f(l)$ is safe to be evaluated, then after execution of p with an input array `inp` containing l the output array `ret` has the elements of $f(l)$. $\mathsf{array}(x, l)$ means that the variable x points to an array that has the same elements as the ones in l.

4. Compiler Verification

Since our compiler (`compileIR`) uses *code templates* for skeletons[1] (Figure 2), the verification process is divided into independent verifications of the templates and the code generator for sequential functions. Below, we first explain the code generation for sequential functions and filling of code templates by showing the compilation process of the skeleton application `map (fun x ⇒ x + 1) arr`. We then present the verification conditions of the code generator and the templates.

Code generator for sequential functions: Given a sequential function, our code generator generates a Coq function (called a code fragment) of type `Var` → (`Cmd` ∗ `Var`), where `Var` and `Cmd` are the types for variables and statements of the language for GPGPU, respectively. The general form of the generated fragments is `fun` $x \Rightarrow (c, r)$, which means a pseudo GPGPU function that returns r as the result after executing c assuming x has the argument value. For example, the code generator generates `fun x ⇒ (l0:=x; l1:=1; l2:=l0+l1, l2)` from the sequential function `fun x ⇒ x + 1`.

Filling holes in templates: Second, the compiler fills a hole in the respective code template with the generated fragment. Code templates are functions taking a code fragment as an argument and generating a GPGPU kernel. We show the *mkMap* template for the `map` skeleton in Figure 3. The notation $t.1$ and $t.2$ are the first and second element of a tuple t, respectively. The Coq expression *mkMap(func)* returns a GPGPU kernel that computes `map f xs` if the array `inp` has an array with elements `xs` and *func* is the generated fragment from `f`. To avoid name conflicts among variables used in generated fragments and code templates, we employ a simple prefix-based convention: all generated variables in fragments are in *GVars*, an infinite set of variables. Currently, *GVars* is the set of variables whose names have the prefix "`l`".

Verification conditions: We manually prove the correctness of each code template in the form of a parameterized Hoare triple with certain assumptions on the parameters (i.e., code fragments). We also prove the correctness of the compiler that generates code fragments from source sequential functions so that the generated code fragments satisfy the abovementioned assumptions. By combining these two proofs, we prove that the whole generated GPGPU kernel satisfies the Hoare triple. Our verification strategy is similar to the one used in the *cross-language linking compiler* [8], in which generated programs by the compiler (in this work, generated fragments from sequential functions) may be linked with hand-written assembly programs (in this work, code templates).

For a sequential function f and its generated fragment *func*, we specify the sequential function compiler correctness: $f \approx_s$ *func* as Definition 1. Then, we specify the correctness of the *mkMap* template as Definition 2.

Definition 1 (The Relation $\approx_s$)**.** *$f \approx_s$ func holds if and only if the following conditions hold.*

- $\forall x : \mathsf{Var}.\ \mathsf{writes}(\mathit{func}(x).1) \subseteq \mathit{GVars}$
- $\forall x : \mathsf{Var}.\ \mathit{func}(x).2 \in \mathit{GVars}$
- $\forall x : \mathsf{Var}.\ x \notin \mathit{GVars} \Rightarrow$
  $\vdash \{x = v\}\, \mathit{func}(x).1\, \{\mathit{func}(x).2 = f(v)\}$
- $\forall x : \mathsf{Var}.\ \mathit{func}(x).1$ *executes no synchronization*

where writes(c) *is the set of written variables in the command c.*

Definition 2 (The specification of *mkMap* template)**.** *For any f, xs and ys, if safe*(`map` *f, xs*) *and $f \approx_s$ func hold, then the following triple holds.*

$$\{\_\mathtt{len} = \mathit{length}(xs) \wedge \mathit{length}(xs) = \mathit{length}(ys) \wedge \mathsf{array}(\mathtt{inp}, xs) \star \mathsf{array}(\mathtt{out}, ys)\}$$
$$\vdash \mathit{mkMap}(\mathit{func})$$
$$\{\mathsf{array}(\mathtt{inp}, xs) \star \mathsf{array}(\mathtt{out}, \mathtt{map}\ f\ xs)\}$$

We also develop a tactic library, namely GPUVeLib, for helping the proof of these correctness. GPUVeLib is based on ones for the semi-automated verification of low-level imperative programs [3].

[1] Our compiler is based on Accelerate [6] with some modification for simplicity. In this paper, we present a more simplified compiler due to limitations of space. Actually, our compiler supports optimizations such as conversion of array of tuples to tuples of arrays and simple fusion transformation.

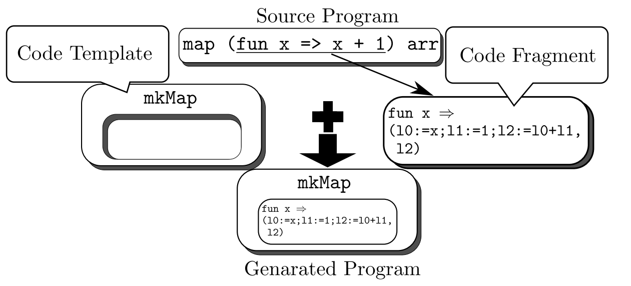

Figure 2. Template-based code generation for skeleton application

```
mkMap(func) :=
  ix := (the thread ID of this thread);
  while (ix < _len) {
    x := inp[ix];
    func(x).1;
    out[ix] := func(x).2;
    ix := ix + (the number of threads); }
```

Figure 3. The *mkMap* code template

5. Current Status and Future Work

Currently, we have already implemented the CertSkel compiler generating GPGPU programs. We have also proved implementations of basic code templates (`map`, `reduce`, etc.) and the code generator for sequential functions. We will verify the entire compiler that also generates host code managing GPU memories and launching GPGPU kernels. We also plan to support more features in other GPUDSLs, such as advanced skeletons (e.g., `scan` and segmented operations) and optimizations (e.g., fusion transformation). Another important feature is *nested parallelism*, which allows skeleton calls inside sequential functions. It is challenging to ensure the correctness of the compilation techniques for nested parallelism (e.g., [2, 4]).

References

[1] I. Asakura, H. Masuhara, and T. Aotani. Proof of Soundness of Concurrent Separation Logic for GPGPU in Coq. *Journal of Information Processing*, 24(1):132–140, 2016.

[2] L. Bergstrom and J. H. Reppy. Nested Data-Parallelism on the GPU. In *ICFP'12*, 2012.

[3] J. Cao, M. Fu, and X. Feng. Practical Tactics for Verifying C Programs in Coq. In *CPP'15*, 2015.

[4] B. C. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *PPOPP'11*, 2011.

[5] X. Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *POPL'06*, 2006.

[6] T. L. McDonell, M. M. T. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP'13*, 2013.

[7] M. Springer and H. Masuhara. Object Support in an Array-Based GPGPU Extension for Ruby. In *ARRAY@PLDI'16*, 2016.

[8] P. Wang, S. Cuellar, and A. Chlipala. Compiler Verification Meets Cross-Language Linking via Data Abstraction. In *OOPSLA'14*, 2014.