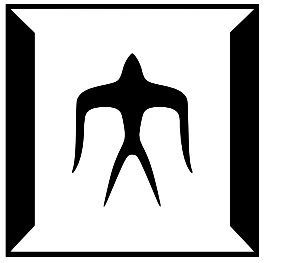


Towards a Formally Verified Skeleton-based Data Parallel DSL for GPGPU

Izumi Asakura, Hidehiko Masuhara, Tomoyuki Aotani
(Tokyo Institute of Technology)



Background & Our Work

	Sequential Lang.	Low-level GPGPU Lang. (e.g., CUDA)	High-level Lang. for GPGPU (e.g., Accelerate [Chakravarty et al. '11] Ikra [Springer et al.'16])
App. level Verification	Many	GPUVerify [Betts et al.'12] PUG [Li et al.'14]	
Verified Compiler	CompCert [Leroy'06] CakeML [Kumar et al.'14]		CertSkel <i>Ours!</i>

Technical Contributions:

- **An Approach to Formally Verified Compiler for Coq-embedded DSLs (compilation by proving)**
- Source code: **a pure Coq function**
- Compiler: **reification** and **verified compiler in pure Coq**
- **Compositional Verification of Template-based Codegen.**
- Compiler specification in Hoare-style logic (we use GPU CSL [Asakura et al.'16], a CSL for GPGPU)
- Compositional verification for each compiler component (similar to Cito [Wang et al.'14], a cross-language linking compiler)

Programming in CertSkel

Source program:
A pure Coq function on lists written in *skeletons*

```
(* argmax.v (a Coq source file) *)
Definition argmax xs :=
  reduce (fun (i,x) (j,y) =>
    if y <= x then (i,x) else (j,y))
    (zip (seq 0 (length xs)) xs)
```

Compilation by proving
Proving a lemma: **there exists a GPGPU program equivalent to the source function**

```
Definition argmaxGPGPU:
  {p : GPGPU.prog | argmax ≈CG p}.
Proof.
  reifyFunc.
  eexists; apply compileOK.
  (* proof of the safety *)
Defined.
```

Compilation by Proving

{p | argmax ≈_{CG} p} → Reflection & Normalization (in Ltac) → reifyFunc → {p | argmaxR ≈_{IG} p} where argmaxR =

Normalized syntax tree of original argmax
Let(int, seq ..., Let(tup(int, int), zip ..., Let(tup(int, int), reduce ..., tmp3)))

eexists; apply compileOK.

Apply the compiler correctness lemma: Lemma compileOK: ∀s, safe(s) ⇒ s ≈_{IG} compileIR s

Defined

Extracting GPGPU program (**CUDA C**) by Coq extraction mechanism
- The extracted OCaml program saves program to "./argmax.cu"

```
Definition res := generateGPGPUFile
  "./argmax.cu" argmaxGPGPU.
Separate Extract res.
```

compileIR: a Certified Template-based Code Generator (written in Coq)

Reified syntax tree of the source function

```
let tmp1 := seq 0 (length xs) in
let tmp2 := zip tmp1 xs in
let tmp3 := reduce (fun (i,x) (j,y) => ...) tmp2 in
tmp3
```

Compositional correctness conditions

func. and GPGPU
 $S \approx_{IG} C \Leftrightarrow$
 $\forall l, s(l) \Downarrow r \Rightarrow$
 $\{array(inp, l)\}$
 $\Gamma \models c(inp)$
 $\{array(inp, l) \star array(res, r)\}$

skel. and template
 $S \approx_T T \Leftrightarrow$
 $\forall f_s f_c. S f_s \mid \Downarrow r \wedge f_s \approx_s f_c \Rightarrow$
 $\{array(inp, l) \star array(res, -)\}$
 $\models T[f_c](res, inp, len, \dots)$
 $\{array(inp, l) \star array(res, r)\}$

seq. func. and code frag.
 $f_s \approx_s f_c \Leftrightarrow$
- $\forall x : Var. writes(f_c(x).1) \subseteq GVars$
- $\forall x : Var. f_c(x).2 \subseteq GVars$
- $\forall x : Var. x \notin GVars \Rightarrow$
 $\models \{x=v\}f_c(x).1 \{f_c(x).2=f_s(v)\}$
- $\forall x : Var. f_s(x).1$ has no barrier

```
T* argmax(...) {
  out1 := alloc(len);
  seqKernel<<<...>>(out1,len,xs);
  out2 := alloc(len);
  zipKernel<<<...>>(out2,len,xs,out1);
  out3 := alloc(nThrd);
  out4 := alloc(1);
  pFoldg1<<<...>>(out3, out1);
  pFoldb1<<<...>>(out4, out2);
  return 3; }
```

A host function (executed on CPU)

Hand-written code template (optimized e.g., AoS to SoA, complex array accessing pattern and sync.)

invokes kernels

Several kernel functions (executed on GPU)

```
(if (y <= x) {res1 = i; res2 = x}
else {res1 = j; res2 = y},
(res1, res2))
```

Code fragment

What We Have Done

- Implemented: all compiler components & simple fusion trans. by rewriting (done before reification)
- Proved: \approx_T for some skels. (map, reduce) and seq. func. compiler
- By using helper tactic library GPUveLib (symbolic execution on statements)

Future Work

- Finish the proof (top level function compilation)
- Precise semantics: machine integer does not have infinite precision
- More optimizations (e.g., fusion transformation)
- More features (e.g., nested parallelism, advanced skeletons)