# Syntax and semantics of a GPU kernel programming language

John Wickerson

March 17, 2025

**Abstract**

This document accompanies the article *The Design and Implementation of a Verification Technique for GPU Kernels* by Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson and John Wickerson [1]. It formalises all of the definitions provided in Sections 3 and 4 of the article.

## Contents



## 1 General purpose definitions and lemmas

**theory** *Misc* **imports**
*Main*
**begin**

A handy abbreviation when working with maps

**abbreviation** *make-map* :: $'a\ set \Rightarrow 'b \Rightarrow ('a \rightharpoonup 'b)$ (‹[ - |=> - ]›)
**where**
$[ks\ |{=}{>}\ v] \equiv \lambda k.\ if\ k \in ks\ then\ Some\ v\ else\ None$

Projecting the components of a triple

**definition** *fst3* ≡ *fst*
**definition** *snd3* ≡ *fst* ∘ *snd*
**definition** *thd3* ≡ *snd* ∘ *snd*

**lemma** *fst3-simp* [*simp*]: *fst3* (*a*,*b*,*c*) = *a* ⟨*proof*⟩
**lemma** *snd3-simp* [*simp*]: *snd3* (*a*,*b*,*c*) = *b* ⟨*proof*⟩
**lemma** *thd3-simp* [*simp*]: *thd3* (*a*,*b*,*c*) = *c* ⟨*proof*⟩

**end**

# 2 Syntax of KPL

**theory** *KPL-syntax* **imports**
*Misc*
**begin**

Locations of local variables

**typedecl** *V*

C strings

**typedecl** *name*

Procedure names

**typedecl** *proc-name*

Local-id, group-id

**type-synonym** *lid* = *nat*
**type-synonym** *gid* = *nat*

Fully-qualified thread-id

**type-synonym** *tid* = *gid* × *lid*

Let $(G, T)$ range over threadsets

**type-synonym** *threadset* = *gid set* × (*gid* ⇀ *lid set*)

Returns the set of tids in a threadset

**fun** *tids* :: *threadset* ⇒ *tid set*
**where**
*tids* (*G*,*T*) = {(*i*,*j*) | *i j*. *i* ∈ *G* ∧ *j* ∈ *the* (*T i*)}

**type-synonym** *word* = *nat*

**datatype** *loc* =
*Name name*
| *Var V*

Local expressions

**datatype** *local-expr* =
  *Loc loc*
| *Gid*
| *Lid*
| *eTrue*
| *eConj local-expr local-expr*  (**infixl** ‹∧∗› *50*)
| *eNot local-expr*  (‹¬∗›)

Basic statements

**datatype** *basic-stmt* =
  *Assign loc local-expr*
| *Read loc local-expr*
| *Write local-expr local-expr*

Statements

**datatype** *stmt* =
  *Basic basic-stmt*
| *Seq stmt stmt* (**infixl** ‹;;› *50*)
| *Local name stmt*
| *If local-expr stmt stmt*
| *While local-expr stmt*
| *WhileDyn local-expr stmt*
| *Call proc-name local-expr*
| *Barrier*
| *Break*
| *Continue*
| *Return*

Procedures comprise a procedure name, parameter name, and a body statement

**record** *proc* =
  *proc-name* :: *proc-name*
  *param* :: *name*
  *body* :: *stmt*

Kernels

**record** *kernel* =
  *groups* :: *nat*
  *threads* :: *nat*
  *procs* :: *proc list*
  *main* :: *stmt*

**end**

# 3 Well-formedness of KPL kernels

**theory** *KPL-wellformedness* **imports**

*KPL-syntax*
**begin**

Well-formed local expressions. *wf-local-expr ns e* means that

- *e* does not mention any internal locations, and
- any name mentioned by *e* is in the set *ns*.

**fun** *wf-local-expr* :: *name set* ⇒ *local-expr* ⇒ *bool*
**where**
  *wf-local-expr ns* (*Loc* (*Var j*)) = *False*
| *wf-local-expr ns* (*Loc* (*Name n*)) = (*n* ∈ *ns*)
| *wf-local-expr ns* (*e1* ∧∗ *e2*) =
  (*wf-local-expr ns e1* ∧ *wf-local-expr ns e2*)
| *wf-local-expr ns* (¬∗ *e*) = *wf-local-expr ns e*
| *wf-local-expr ns* - = *True*

Well-formed basic statements. *wf-basic-stmt ns b* means that

- *b* does not mention any internal locations, and
- any name mentioned by *b* is in the set *ns*.

**fun** *wf-basic-stmt* :: *name set* ⇒ *basic-stmt* ⇒ *bool*
**where**
  *wf-basic-stmt ns* (*Assign x e*) = *wf-local-expr ns e*
| *wf-basic-stmt ns* (*Read x e*) = *wf-local-expr ns e*
| *wf-basic-stmt ns* (*Write e1 e2*) =
  (*wf-local-expr ns e1* ∧ *wf-local-expr ns e2*)

Well-formed statements. *wf-stmt ns F S* means:

- *S* only calls procedures whose name is in *F*,
- *S* does not contain *WhileDyn*,
- *S* does not mention internal variables,
- *S* only mentions names in *ns*, and
- *S* does not declare the same name twice, e.g. *Local x* (*Local x foo*).

**fun** *wf-stmt* :: *name set* ⇒ *proc-name set* ⇒ *stmt* ⇒ *bool*
**where**
  *wf-stmt ns F* (*Basic b*) = *wf-basic-stmt ns b*
| *wf-stmt ns F* (*S1* ;; *S2*) = (*wf-stmt ns F S1* ∧ *wf-stmt ns F S2*)
| *wf-stmt ns F* (*Local n S*) = (*n* ∉ *ns* ∧ *wf-stmt* ({*n*} ∪ *ns*) *F S*)
| *wf-stmt ns F* (*If e S1 S2*) =
  (*wf-local-expr ns e* ∧ *wf-stmt ns F S1* ∧ *wf-stmt ns F S2*)

| *wf-stmt ns F* (*While e S*) =
(*wf-local-expr ns e* ∧ *wf-stmt ns F S*)
| *wf-stmt ns F* (*WhileDyn - -*) = *False*
| *wf-stmt ns F* (*Call f e*) = (*f* ∈ *F* ∧ *wf-local-expr ns e*)
| *wf-stmt - - -* = *True*

*no-return S* holds if *S* does not contain a *Return* statement

**fun** *no-return* :: *stmt* ⇒ *bool*
**where**
*no-return* (*S1* ;; *S2*) = (*no-return S1* ∧ *no-return S2*)
| *no-return* (*Local n S*) = *no-return S*
| *no-return* (*If e S1 S2*) = (*no-return S1* ∧ *no-return S2*)
| *no-return* (*While e S*) = (*no-return S*)
| *no-return Return* = *False*
| *no-return -* = *True*

Well-formed kernel

**definition** *wf-kernel* :: *kernel* ⇒ *bool*
**where**
*wf-kernel P* ≡
*let F* = *set* (*map proc-name* (*procs P*)) *in*

— The main statement must not refer to *any* variable, except those it locally defines.
*wf-stmt* {} *F* (*main P*)

— The main statement contains no return statement.
∧ *no-return* (*main P*)

— A procedure body may refer only to its argument.
∧ *list-all* (λ*f*. *wf-stmt* {*param f*} *F* (*body f*)) (*procs P*)

**end**

## 4 Thread, group and kernel states

**theory** *KPL-state* **imports**
*KPL-syntax*
**begin**

Thread state

**record** *thread-state* =

*l* :: *V* + *bool* ⇒ *word*
*sh* :: *nat* ⇒ *word*
*R* :: *nat set*
*W* :: *nat set*

**abbreviation** *GID* ≡ *Inr True*
**abbreviation** *LID* ≡ *Inr False*

Group state

**record** *group-state* =
  *thread-states* :: *lid* ⇀ *thread-state* (‹- $_{ts}$› [*1000*] *1000*)
  *R-group* :: (*lid* × *nat*) *set*
  *W-group* :: (*lid* × *nat*) *set*

Valid group state

**fun** *valid-group-state* :: (*gid* ⇀ *lid set*) ⇒ *gid* ⇒ *group-state* ⇒ *bool*
**where**
  *valid-group-state T i* γ = (
  *dom* (γ $_{ts}$) = *the* (*T i*) ∧
  (∀ *j* ∈ *the* (*T i*).
  *l* (*the* (γ $_{ts}$ *j*)) *GID* = *i* ∧
  *l* (*the* (γ $_{ts}$ *j*)) *LID* = *j*))

Predicated statements

**type-synonym** *pred-stmt* = *stmt* × *local-expr*
**type-synonym** *pred-basic-stmt* = *basic-stmt* × *local-expr*

Kernel state

**type-synonym** *kernel-state* =
  (*gid* ⇀ *group-state*) × *pred-stmt list* × *V list*

Valid kernel state

**fun** *valid-kernel-state* :: *threadset* ⇒ *kernel-state* ⇒ *bool*
**where**
  *valid-kernel-state* (*G*,*T*) (κ, *ss*, -) = (
  *dom* κ = *G* ∧
  (∀ *i* ∈ *G*. *valid-group-state T i* (*the* (κ *i*))))

Valid initial kernel state

**fun** *valid-initial-kernel-state* :: *stmt* ⇒ *threadset* ⇒ *kernel-state* ⇒ *bool*
**where**
  *valid-initial-kernel-state S* (*G*,*T*) (κ, *ss*, *vs*) = (
  *valid-kernel-state* (*G*,*T*) (κ, *ss*, *vs*) ∧
  (*ss* = [(*S*, *eTrue*)]) ∧
  (∀ *i* ∈ *G*. *R-group* (*the* (κ *i*)) = {} ∧ *W-group* (*the* (κ *i*)) = {}) ∧
  (∀ *i* ∈ *G*. ∀ *j* ∈ *the* (*T i*). *R* (*the* ((*the* (κ *i*))$_{ts}$ *j*)) = {}
    ∧ *W* (*the* ((*the* (κ *i*))$_{ts}$ *j*)) = {}) ∧
  (∀ *i* ∈ *G*. ∀ *j* ∈ *the* (*T i*). ∀ *v* :: *V*.
    *l* (*the* ((*the* (κ *i*))$_{ts}$ *j*)) (*Inl v*) = *0*) ∧
  (∀ *i* ∈ *G*. ∀ *i'* ∈ *G*. ∀ *j* ∈ *the* (*T i*). ∀ *j'* ∈ *the* (*T i'*).
    *sh* (*the* ((*the* (κ *i*))$_{ts}$ *j*)) =
    *sh* (*the* ((*the* (κ *i'*))$_{ts}$ *j'*))) ∧

(*vs* = []))

**end**

# 5 Execution rules for threads

**theory** *KPL-execution-thread* **imports**
*KPL-state*
**begin**

Evaluate a local expression down to a word

**fun** $eval\text{-}word :: local\text{-}expr \Rightarrow thread\text{-}state \Rightarrow word$
**where**
$eval\text{-}word\ (Loc\ (Var\ v))\ \tau = l\ \tau\ (Inl\ v)$

$|\ eval\text{-}word\ Lid\ \tau = l\ \tau\ LID$
$|\ eval\text{-}word\ Gid\ \tau = l\ \tau\ GID$
$|\ eval\text{-}word\ eTrue\ \tau = 1$
$|\ eval\text{-}word\ (e1\ \wedge{*}\ e2)\ \tau =$
$(eval\text{-}word\ e1\ \tau * eval\text{-}word\ e2\ \tau)$
$|\ eval\text{-}word\ (\neg{*}\ e)\ \tau = (if\ eval\text{-}word\ e\ \tau = 0\ then\ 1\ else\ 0)$

Evaluate a local expression down to a boolean

**fun** $eval\text{-}bool :: local\text{-}expr \Rightarrow thread\text{-}state \Rightarrow bool$
**where**
$eval\text{-}bool\ e\ \tau = (eval\text{-}word\ e\ \tau \neq 0)$

Abstraction level: none, equality abstraction, or adversarial abstraction

**datatype** $abs\text{-}level = No\text{-}Abst\ |\ Eq\text{-}Abst\ |\ Adv\text{-}Abst$

The rules of Figure 4, plus two additional rules for adversarial abstraction (Fig 7b)

**inductive** *step-t*
$:: abs\text{-}level \Rightarrow (thread\text{-}state \times pred\text{-}basic\text{-}stmt) \Rightarrow thread\text{-}state \Rightarrow bool$
**where**
*T-Disabled*:
$\neg\ (eval\text{-}bool\ p\ \tau) \Longrightarrow step\text{-}t\ a\ (\tau,\ (b,\ p))\ \tau$
| *T-Assign*:
$[\![\ eval\text{-}bool\ p\ \tau\ ;\ l' = (l\ \tau)\ (Inl\ v := eval\text{-}word\ e\ \tau)\ ]\!]$
$\Longrightarrow step\text{-}t\ a\ (\tau,\ (Assign\ (Var\ v)\ e,\ p))\ (\tau\ (|\ l := l'\ |))$
| *T-Read*:
$[\![\ eval\text{-}bool\ p\ \tau\ ;\ l' = (l\ \tau)\ (Inl\ v := sh\ \tau\ (eval\text{-}word\ e\ \tau))\ ;$
$R' = R\ \tau \cup \{\ eval\text{-}word\ e\ \tau\ \}\ ;\ a \in \{No\text{-}Abst,\ Eq\text{-}Abst\}\ ]\!]$
$\Longrightarrow step\text{-}t\ a\ (\tau,\ (Read\ (Var\ v)\ e,\ p))\ (\tau\ (|\ l := l',\ R := R'\ |))$
| *T-Write*:
$[\![\ eval\text{-}bool\ p\ \tau\ ;$
$sh' = (sh\ \tau)\ (eval\text{-}word\ e1\ \tau := eval\text{-}word\ e2\ \tau)\ ;$
$W' = W\ \tau \cup \{\ eval\text{-}word\ e1\ \tau\ \}\ ;\ a \in \{No\text{-}Abst,\ Eq\text{-}Abst\}\ ]\!]$

⟹ step-t a (τ, (Write e1 e2, p)) (τ (| sh := sh′, W := W′ |))
| T-Read-Adv:
⟦ eval-bool p τ ; l′ = (l τ) (Inl v := asterisk) ;
R′ = R τ ∪ { eval-word e τ } ⟧
⟹ step-t Adv-Abst (τ, (Read (Var v) e, p)) (τ (| l := l′, R := R′ |))
| T-Write-Adv:
⟦ eval-bool p τ ; W′ = W τ ∪ { eval-word e1 τ } ⟧
⟹ step-t Adv-Abst (τ, (Write e1 e2, p)) (τ (| ~~sh := sh′,~~ W := W′ |))

Rephrasing *T-Assign* to make it more usable

**lemma** *T-Assign-helper*:
⟦ eval-bool p τ ; l′ = (l τ) (Inl v := eval-word e τ) ; τ′ = τ (| l := l′ |) ⟧
⟹ step-t a (τ, (Assign (Var v) e, p)) τ′
⟨proof⟩

Rephrasing *T-Read* to make it more usable

**lemma** *T-Read-helper*:
⟦ eval-bool p τ ; l′ = (l τ) (Inl v := sh τ (eval-word e τ)) ;
R′ = R τ ∪ { eval-word e τ } ; a ∈ {No-Abst, Eq-Abst} ;
τ′ = τ (| l := l′, R := R′ |) ⟧
⟹ step-t a (τ, (Read (Var v) e, p)) τ′
⟨proof⟩

Rephrasing *T-Write* to make it more usable

**lemma** *T-Write-helper*:
⟦ eval-bool p τ ;
sh′ = (sh τ) (eval-word e1 τ := eval-word e2 τ) ;
W′ = W τ ∪ { eval-word e1 τ } ; a ∈ {No-Abst, Eq-Abst} ;
τ′ = τ (| sh := sh′, W := W′ |) ⟧
⟹ step-t a (τ, (Write e1 e2, p)) τ′
⟨proof⟩

**end**

# 6 Execution rules for groups

**theory** *KPL-execution-group* **imports**
*KPL-execution-thread*
**begin**

Intra-group race detection

**definition** *group-race*
:: lid set ⇒ (lid ⇀ thread-state) ⇒ bool
**where** group-race T γ ≡
∃ j ∈ T. ∃ k ∈ T. j ≠ k ∧
W (the (γ j)) ∩ (R (the (γ k)) ∪ W (the (γ k))) ≠ {}

The constraints for the *merge* map

**inductive** *pre-merge*
$:: lid\ set \Rightarrow (lid \rightharpoonup thread\text{-}state) \Rightarrow nat \Rightarrow word \Rightarrow bool$
**where**
$[\![\ j \in T\ ;\ z \in W\ (the\ (\gamma\ j))\ ;\ dom\ \gamma = T\ ]\!] \Longrightarrow$
$pre\text{-}merge\ T\ \gamma\ z\ (sh\ (the\ (\gamma\ j))\ z)$
$|\ [\![\ \forall j \in T.\ z \notin W\ (the\ (\gamma\ j))\ ;\ dom\ \gamma = T\ ]\!] \Longrightarrow$
$pre\text{-}merge\ T\ \gamma\ z\ (sh\ (the\ (\gamma\ 0))\ z)$

**inductive-cases** *pre-merge-inv* [*elim*!]: $pre\text{-}merge\ P\ \gamma\ z\ z'$

The *merge* map maps each nat to the word that satisfies the above constaints. The *merge-is-unique* lemma shows that there exists exactly one such word per nat, provided there are no group races.

**definition** $merge :: lid\ set \Rightarrow (lid \rightharpoonup thread\text{-}state) \Rightarrow nat \Rightarrow word$
**where** $merge\ T\ \gamma \equiv \lambda z.\ The\ (pre\text{-}merge\ T\ \gamma\ z)$

**lemma** *no-races-imp-no-write-overlap*:
$\neg\ (group\text{-}race\ T\ \gamma) \Longrightarrow$
$\forall i \in T.\ \forall j \in T.$
$i \neq j \longrightarrow W\ (the\ (\gamma\ i)) \cap W\ (the\ (\gamma\ j)) = \{\}$
$\langle proof \rangle$

**lemma** *merge-is-unique*:
**assumes** $dom\ \gamma = T$
**assumes** $\neg\ (group\text{-}race\ T\ \gamma)$
**shows** $\exists! z'.\ pre\text{-}merge\ T\ \gamma\ z\ z'$
$\langle proof \rangle$

The rules of Figure 5, plus an additional rule for equality abstraction (Fig 7a), plus an additional rule for adversarial abstraction (Fig 7b)

**inductive** *step-g*
$:: abs\text{-}level \Rightarrow gid \Rightarrow (gid \rightharpoonup lid\ set) \Rightarrow (group\text{-}state \times pred\text{-}stmt) \Rightarrow group\text{-}state\ option \Rightarrow bool$
**where**
*G-Race*:
$[\![\ \forall j \in the\ (T\ i).\ step\text{-}t\ a\ (the\ (\gamma\ _{ts}\ j),\ (s,\ p))\ (the\ (\gamma'\ _{ts}\ j))\ ;$
$group\text{-}race\ (the\ (T\ i))\ ((\gamma' :: group\text{-}state)_{ts})\ ]\!]$
$\Longrightarrow step\text{-}g\ a\ i\ T\ (\gamma,\ (Basic\ s,\ p))\ None$
| *G-Basic*:
$[\![\ \forall j \in the\ (T\ i).\ step\text{-}t\ a\ (the\ (\gamma\ _{ts}\ j),\ (s,\ p))\ (the\ (\gamma'\ _{ts}\ j))\ ;$
$\neg\ (group\text{-}race\ (the\ (T\ i))\ (\gamma'\ _{ts}))\ ;$
$R\text{-}group\ \gamma' = R\text{-}group\ \gamma \cup (\bigcup j \in the\ (T\ i).\ (\{j\} \times R\ (the\ (\gamma'\ _{ts}\ j))))\ ;$
$W\text{-}group\ \gamma' = W\text{-}group\ \gamma \cup (\bigcup j \in the\ (T\ i).\ (\{j\} \times W\ (the\ (\gamma'\ _{ts}\ j))))\ ]\!]$
$\Longrightarrow step\text{-}g\ a\ i\ T\ (\gamma,\ (Basic\ s,\ p))\ (Some\ \gamma')$
| *G-No-Op*:
$\forall j \in the\ (T\ i).\ \neg\ (eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j)))$
$\Longrightarrow step\text{-}g\ a\ i\ T\ (\gamma,\ (Barrier,\ p))\ (Some\ \gamma)$
| *G-Divergence*:
$[\![\ j \neq k\ ;\ j \in the\ (T\ i)\ ;\ k \in the\ (T\ i)\ ;$

$eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j))\ ;\ \neg\ (eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ k)))\ \rrbracket$
$\Longrightarrow step\text{-}g\ a\ i\ T\ (\gamma,\ (Barrier,\ p))\ None$
$|\ G\text{-}Sync$:
$\llbracket\ \forall j \in the\ (T\ i).\ eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j))\ ;$
$\forall j \in the\ (T\ i).\ the\ (\gamma'\ _{ts}\ j) = (the\ (\gamma\ _{ts}\ j))\ (|$
$sh := merge\ P\ (\gamma\ _{ts}),\ R := \{\},\ W := \{\}\ |)\ \rrbracket$
$\Longrightarrow step\text{-}g\ No\text{-}Abst\ i\ T\ (\gamma,\ (Barrier,\ p))\ (Some\ \gamma')$
$|\ G\text{-}Sync\text{-}Eq$:
$\llbracket\ \forall j \in the\ (T\ i).\ eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j))\ ;$
$\forall j \in the\ (T\ i).\ the\ (\gamma'\ _{ts}\ j) = (the\ (\gamma\ _{ts}\ j))\ (|$
$sh := sh',\ R := \{\},\ W := \{\}\ |)\ \rrbracket$
$\Longrightarrow step\text{-}g\ Eq\text{-}Abst\ i\ T\ (\gamma,\ (Barrier,\ p))\ (Some\ \gamma')$
$|\ G\text{-}Sync\text{-}Adv$:
$\llbracket\ \forall j \in the\ (T\ i).\ eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j))\ ;$
$\forall j \in the\ (T\ i).\ \exists sh'.\ the\ (\gamma'\ _{ts}\ j) = (the\ (\gamma\ _{ts}\ j))\ (|$
$sh := sh',\ R := \{\},\ W := \{\}\ |)\ \rrbracket$
$\Longrightarrow step\text{-}g\ Adv\text{-}Abst\ i\ T\ (\gamma,\ (Barrier,\ p))\ (Some\ \gamma')$

Rephrasing *G-No-Op* to make it more usable

**lemma** *G-No-Op-helper*:
$\llbracket\ \forall j \in the\ (T\ i).\ \neg\ (eval\text{-}bool\ p\ (the\ (\gamma\ _{ts}\ j)))\ ;\ \gamma = \gamma'\ \rrbracket$
$\Longrightarrow step\text{-}g\ a\ i\ T\ (\gamma,\ (Barrier,\ p))\ (Some\ \gamma')$
$\langle proof\rangle$

**end**

# 7 Execution rules for kernels

**theory** *KPL-execution-kernel* **imports**
*KPL-execution-group*
**begin**

Inter-group race detection

**definition** *kernel-race*
$::\ gid\ set \Rightarrow (gid \rightharpoonup group\text{-}state) \Rightarrow bool$
**where** $kernel\text{-}race\ G\ \kappa \equiv$
$\exists i \in G.\ \exists j \in G.\ i \neq j\ \wedge$
$(snd\ `\ (W\text{-}group\ (the\ (\kappa\ i))))\ \cap$
$(snd\ `\ (R\text{-}group\ (the\ (\kappa\ j)))\ \cup\ snd\ `\ (W\text{-}group\ (the\ (\kappa\ j))))\ \neq \{\}$

Replaces top-level *Break* with $v := true$

**fun** $belim :: stmt \Rightarrow V \Rightarrow stmt$
**where**
$belim\ (Basic\ b)\ v = Basic\ b$
$|\ belim\ (S1\ ;;\ S2)\ v = (belim\ S1\ v\ ;;\ belim\ S2\ v)$
$|\ belim\ (Local\ n\ S)\ v = Local\ n\ (belim\ S\ v)$
$|\ belim\ (If\ e\ S1\ S2)\ v = If\ e\ (belim\ S1\ v)\ (belim\ S2\ v)$

| *belim* (*While e S*) *v* = *While e S*

| *belim* (*Call f e*) *v* = *Call f e*
| *belim Barrier v* = *Barrier*
| *belim Break v* = *Basic* (*Assign* (*Var v*) *eTrue*)
| *belim Continue v* = *Continue*
| *belim Return v* = *Return*

Replaces top-level *Continue* with *v* := *true*

**fun** *celim* :: *stmt* ⇒ *V* ⇒ *stmt*
**where**
*celim* (*Basic b*) *v* = *Basic b*
| *celim* (*S1* ;; *S2*) *v* = (*celim S1 v* ;; *celim S2 v*)
| *celim* (*Local n S*) *v* = *Local n* (*celim S v*)
| *celim* (*If e S1 S2*) *v* = *If e* (*celim S1 v*) (*celim S2 v*)
| *celim* (*While e S*) *v* = *While e S*

| *celim* (*Call f e*) *v* = *Call f e*
| *celim Barrier v* = *Barrier*
| *celim Break v* = *Break*
| *celim Continue v* = *Basic* (*Assign* (*Var v*) *eTrue*)
| *celim Return v* = *Return*

*subst-basic-stmt n v loc* replaces *n* with *v* inside *loc*

**fun** *subst-loc* :: *name* ⇒ *V* ⇒ *loc* ⇒ *loc*
**where**
*subst-loc n v* (*Var w*) = *Var w*
| *subst-loc n v* (*Name m*) = (*if n* = *m then Var v else Name m*)

*subst-local-expr n v e* replaces *n* with *v* inside *e*

**fun** *subst-local-expr*
:: *name* ⇒ *V* ⇒ *local-expr* ⇒ *local-expr*
**where**
*subst-local-expr n v* (*Loc loc*) = *Loc* (*subst-loc n v loc*)
| *subst-local-expr n v Gid* = *Gid*
| *subst-local-expr n v Lid* = *Lid*
| *subst-local-expr n v eTrue* = *eTrue*
| *subst-local-expr n v* (*e1* ∧∗ *e2*) =
(*subst-local-expr n v e1* ∧∗ *subst-local-expr n v e2*)
| *subst-local-expr n v* (¬∗ *e*) = ¬∗ (*subst-local-expr n v e*)

*subst-basic-stmt n v b* replaces *n* with *v* inside *b*

**fun** *subst-basic-stmt* :: *name* ⇒ *V* ⇒ *basic-stmt* ⇒ *basic-stmt*
**where**
*subst-basic-stmt n v* (*Assign loc e*) =
*Assign* (*subst-loc n v loc*) (*subst-local-expr n v e*)
| *subst-basic-stmt n v* (*Read loc e*) =
*Read* (*subst-loc n v loc*) (*subst-local-expr n v e*)

| *subst-basic-stmt n v* (*Write e1 e2*) =
  *Write* (*subst-local-expr n v e1*) (*subst-local-expr n v e2*)

*subst-stmt n v s t* holds if $t$ is the result of replacing $n$ with $v$ inside $s$

**inductive** *subst-stmt* :: *name* ⇒ *V* ⇒ *stmt* ⇒ *stmt* ⇒ *bool*
**where**
  *subst-stmt n v* (*Basic b*) (*Basic* (*subst-basic-stmt n v b*))
| ⟦ *subst-stmt n v S1 S1′* ; *subst-stmt n v S2 S2′* ⟧ ⟹
  *subst-stmt n v* (*S1* ;; *S2*) (*S1′* ;; *S2′*)
| ⟦ *m* ≠ *n* ; *subst-stmt n v S S′* ⟧ ⟹
  *subst-stmt n v* (*Local m S*) (*Local m S′*)
| ⟦ *subst-stmt n v S1 S1′* ; *subst-stmt n v S2 S2′* ⟧ ⟹
  *subst-stmt n v* (*If e S1 S2*) (*If e S1′ S2′*)
| *subst-stmt n v S S′* ⟹ *subst-stmt n v* (*While e S*) (*While e S′*)

| *subst-stmt n v* (*Call f e*) (*Call f e*)
| *subst-stmt n v Barrier Barrier*
| *subst-stmt n v Break Break*
| *subst-stmt n v Continue Continue*
| *subst-stmt n v Return Return*

*param-subst f u* replaces $f$'s parameter with $u$

**definition** *param-subst* :: *proc list* ⇒ *proc-name* ⇒ *V* ⇒ *stmt*
**where** *param-subst fs f u* ≡
  *let proc* = *THE proc. proc* ∈ *set fs* ∧ *proc-name proc* = *f in*
  *THE S′. subst-stmt* (*param proc*) *u* (*body proc*) *S′*

Replace *Return* with $v$ := *true*

**fun** *relim* :: *stmt* ⇒ *V* ⇒ *stmt*
**where**
  *relim* (*Basic b*) *v* = *Basic b*

| *relim* (*S1* ;; *S2*) *v* = (*relim S1 v* ;; *relim S2 v*)
| *relim* (*Local n S*) *v* = *Local n* (*relim S v*)
| *relim* (*If e S1 S2*) *v* = *If e* (*relim S1 v*) (*relim S2 v*)
| *relim* (*While e S*) *v* = *While e* (*relim S v*)

| *relim* (*Call f e*) *v* = *Call f e*
| *relim Barrier v* = *Barrier*
| *relim Break v* = *Break*
| *relim Continue v* = *Continue*
| *relim Return v* = *Basic* (*Assign* (*Var v*) *eTrue*)

Fresh variables

**definition** *fresh* :: *V* ⇒ *V list* ⇒ *bool*
**where** *fresh v vs* ≡ *v* ∉ *set vs*

The rules of Figure 6

**inductive** *step-k*
:: *abs-level* ⇒ *proc list* ⇒ *threadset* ⇒ *kernel-state* ⇒ *kernel-state option* ⇒ *bool*
**where**
*K-Inter-Group-Race*:
⟦ ∀ i ∈ G. step-g a i T (the (κ i), (Basic b, p)) (Some (the (κ′ i))) ;
kernel-race P κ′ ⟧ ⟹
step-k a fs (G,T) (κ, (Basic b, p) # ss, vs) None
| *K-Intra-Group-Race*:
⟦ i ∈ G; step-g a i T (the (κ i), (Basic s, p)) None ⟧ ⟹
step-k a fs (G,T) (κ, (Basic s, p) # ss, vs) None
| *K-Basic*:
⟦ ∀ i ∈ G. step-g a i T (the (κ i), (Basic b, p)) (Some (the (κ′ i))) ;
¬ (kernel-race G κ′) ⟧ ⟹
step-k a fs (G,T) (κ, (Basic b, p) # ss, vs) (Some (κ′,ss, vs))
| *K-Divergence*:
⟦ i ∈ G; step-g a i T (the (κ i), (Barrier, p)) None ⟧ ⟹
step-k a fs (G,T) (κ, (Barrier, p) # ss, vs) None
| *K-Sync*:
⟦ ∀ i ∈ G. step-g a i T (the (κ i), (Barrier, p)) (Some (the (κ′ i))) ;
¬ (kernel-race G κ′) ⟧ ⟹
step-k a fs (G,T) (κ, (Barrier, p) # ss, vs) (Some (κ′,ss, vs))
| *K-Seq*:
step-k a fs (G,T) (κ, (S1 ;; S2, p) # ss, vs)
(Some (κ, (S1, p) # (S2, p) # ss, vs))
| *K-Var*:
fresh v vs ⟹
step-k a fs (G,T) (κ, (Local n S, p) # ss, vs)
(Some (κ, (THE S′. subst-stmt n v S S′, p) # ss, v # vs))
| *K-If*:
fresh v vs ⟹
step-k a fs (G,T) (κ, (If e S1 S2, p) # ss, vs) (Some (κ,
(Basic (Assign (Var v) e), p)
# (S1, p ∧∗ Loc (Var v))
# (S2, p ∧∗ ¬∗ (Loc (Var v))) # ss, v # vs))
| *K-Open*:
fresh v vs ⟹
step-k a fs (G,T) (κ, (While e S, p) # ss, vs) (Some (κ,
(WhileDyn e (belim S v), p ∧∗ ¬∗ (Loc (Var v))) # ss, v # vs))
| *K-Iter*:
⟦ i ∈ G ; j ∈ the (T i) ;
eval-bool (p ∧∗ e) (the ((the (κ i))$_{ts}$ j)) ;
fresh u vs ; fresh v vs; u ≠ v ⟧ ⟹
step-k a fs (G,T) (κ, (WhileDyn e S, p) # ss, vs) (Some (κ,
(Basic (Assign (Var u) e), p)
# (celim S v, p ∧∗ Loc (Var u) ∧∗ ¬∗ (Loc (Var v)))
# (WhileDyn e S, p) # ss, u # v # vs))
| *K-Done*:
∀ i ∈ G. ∀ j ∈ the (T i).
¬ (eval-bool (p ∧∗ e) (the ((the (κ i))$_{ts}$ j))) ⟹

*step-k a fs* $(G,T)$ $(\kappa,$ (*WhileDyn e S*, *p*) # *ss*, *vs*) (*Some* $(\kappa,$ *ss*, *vs*))
| *K-Call*:
⟦ *fresh u vs* ; *fresh v vs* ; $u \neq v$ ; *s* = *param-subst fs f u* ⟧ $\Longrightarrow$
*step-k a fs* $(G,T)$ $(\kappa,$ (*Call f e*, *p*) # *ss*, *vs* )
(*Some* $(\kappa,$ (*Basic* (*Assign* (*Var u*) *e*) ;; *relim s v*,
*p* $\wedge*$ $\neg*$ (*Loc* (*Var v*))) # *ss*, *u* # *v* # *vs*))

**end**

**theory** *Kernel-programming-language* **imports**

*Misc*

*KPL-syntax*

*KPL-wellformedness*

*KPL-state*

*KPL-execution-thread*

*KPL-execution-group*

*KPL-execution-kernel*
**begin**

**end**

# References

[1] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels, 2014. Under submission.