

Syntax and semantics of a GPU kernel programming language

John Wickerson

April 11, 2026

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	1
2	Syntax of KPL	2
3	Well-formedness of KPL kernels	3
4	Thread, group and kernel states	5
5	Execution rules for threads	7
6	Execution rules for groups	8
7	Execution rules for kernels	11

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 \rightarrow 'b) (\langle [- | \Rightarrow -] \rangle)

where

[*ks* | \Rightarrow *v*] \equiv λk . if *k* \in *ks* then *Some v* else *None*

Projecting the components of a triple

definition $fst3 \equiv fst$

definition $snd3 \equiv fst \circ snd$

definition $thd3 \equiv snd \circ snd$

lemma $fst3\text{-simp}$ [*simp*]: $fst3 (a,b,c) = a$ **by** (*simp add: fst3-def*)

lemma $snd3\text{-simp}$ [*simp*]: $snd3 (a,b,c) = b$ **by** (*simp add: snd3-def*)

lemma $thd3\text{-simp}$ [*simp*]: $thd3 (a,b,c) = c$ **by** (*simp add: thd3-def*)

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

$tids (G,T) = \{(i,j) \mid i \ j. i \in G \wedge j \in the (T i)\}$

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. $wf\text{-local}\text{-expr } ns \ e$ means that

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

fun $wf\text{-local}\text{-expr} :: name \ set \Rightarrow local\text{-expr} \Rightarrow bool$

where

$wf\text{-local}\text{-expr } ns \ (Loc \ (Var \ j)) = False$
| $wf\text{-local}\text{-expr } ns \ (Loc \ (Name \ n)) = (n \in ns)$
| $wf\text{-local}\text{-expr } ns \ (e1 \ \wedge^* \ e2) =$
 $(wf\text{-local}\text{-expr } ns \ e1 \ \wedge \ wf\text{-local}\text{-expr } ns \ e2)$
| $wf\text{-local}\text{-expr } ns \ (\neg^* \ e) = wf\text{-local}\text{-expr } ns \ e$
| $wf\text{-local}\text{-expr } ns \ - = True$

Well-formed basic statements. $wf\text{-basic}\text{-stmt } ns \ b$ means that

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

fun $wf\text{-basic}\text{-stmt} :: name \ set \Rightarrow basic\text{-stmt} \Rightarrow bool$

where

$wf\text{-basic}\text{-stmt } ns \ (Assign \ x \ e) = wf\text{-local}\text{-expr } ns \ e$
| $wf\text{-basic}\text{-stmt } ns \ (Read \ x \ e) = wf\text{-local}\text{-expr } ns \ e$
| $wf\text{-basic}\text{-stmt } ns \ (Write \ e1 \ e2) =$
 $(wf\text{-local}\text{-expr } ns \ e1 \ \wedge \ wf\text{-local}\text{-expr } ns \ e2)$

Well-formed statements. $wf\text{-stmt } ns \ F \ S$ 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 $wf\text{-stmt} :: name \ set \Rightarrow proc\text{-name} \ set \Rightarrow stmt \Rightarrow bool$

where

$wf\text{-stmt } ns \ F \ (Basic \ b) = wf\text{-basic}\text{-stmt } ns \ b$
| $wf\text{-stmt } ns \ F \ (S1 \ ;; \ S2) = (wf\text{-stmt } ns \ F \ S1 \ \wedge \ wf\text{-stmt } ns \ F \ S2)$
| $wf\text{-stmt } ns \ F \ (Local \ n \ S) = (n \notin ns \ \wedge \ wf\text{-stmt } (\{n\} \cup ns) \ F \ S)$
| $wf\text{-stmt } ns \ F \ (If \ e \ S1 \ S2) =$
 $(wf\text{-local}\text{-expr } ns \ e \ \wedge \ wf\text{-stmt } ns \ F \ S1 \ \wedge \ wf\text{-stmt } ns \ F \ S2)$

```

| 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.
 \wedge *no-return* (main P)

— A procedure body may refer only to its argument.
 \wedge *list-all* (λf . 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 $GID \equiv \text{Inr True}$

abbreviation $LID \equiv \text{Inr False}$

Group state

record $\text{group-state} =$
 $\text{thread-states} :: \text{lid} \rightarrow \text{thread-state } (\langle _ _ \rangle [1000] 1000)$
 $R\text{-group} :: (\text{lid} \times \text{nat}) \text{ set}$
 $W\text{-group} :: (\text{lid} \times \text{nat}) \text{ set}$

Valid group state

fun $\text{valid-group-state} :: (\text{gid} \rightarrow \text{lid set}) \Rightarrow \text{gid} \Rightarrow \text{group-state} \Rightarrow \text{bool}$
where

$\text{valid-group-state } T i \gamma = ($\text{dom } (\gamma \text{ }_{ts}) = \text{the } (T i) \wedge$
 $(\forall j \in \text{the } (T i).$
 $l (\text{the } (\gamma \text{ }_{ts} j)) \text{ } GID = i \wedge$
 $l (\text{the } (\gamma \text{ }_{ts} j)) \text{ } LID = j))$$

Predicated statements

type-synonym $\text{pred-stmt} = \text{stmt} \times \text{local-expr}$

type-synonym $\text{pred-basic-stmt} = \text{basic-stmt} \times \text{local-expr}$

Kernel state

type-synonym $\text{kernel-state} =$
 $(\text{gid} \rightarrow \text{group-state}) \times \text{pred-stmt list} \times V \text{ list}$

Valid kernel state

fun $\text{valid-kernel-state} :: \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{bool}$

where

$\text{valid-kernel-state } (G, T) (\kappa, \text{ss}, -) = ($\text{dom } \kappa = G \wedge$
 $(\forall i \in G. \text{valid-group-state } T i (\text{the } (\kappa i))))$$

Valid initial kernel state

fun $\text{valid-initial-kernel-state} :: \text{stmt} \Rightarrow \text{threadset} \Rightarrow \text{kernel-state} \Rightarrow \text{bool}$

where

$\text{valid-initial-kernel-state } S (G, T) (\kappa, \text{ss}, \text{vs}) = ($\text{valid-kernel-state } (G, T) (\kappa, \text{ss}, \text{vs}) \wedge$
 $(\text{ss} = [(S, \text{eTrue}]]) \wedge$
 $(\forall i \in G. R\text{-group } (\text{the } (\kappa i)) = \{\}) \wedge W\text{-group } (\text{the } (\kappa i)) = \{\}) \wedge$
 $(\forall i \in G. \forall j \in \text{the } (T i). R (\text{the } ((\text{the } (\kappa i))_{ts} j)) = \{\}$
 $\wedge W (\text{the } ((\text{the } (\kappa i))_{ts} j)) = \{\}) \wedge$
 $(\forall i \in G. \forall j \in \text{the } (T i). \forall v :: V.$
 $l (\text{the } ((\text{the } (\kappa i))_{ts} j)) (\text{Inl } v) = 0) \wedge$
 $(\forall i \in G. \forall i' \in G. \forall j \in \text{the } (T i). \forall j' \in \text{the } (T i').$
 $sh (\text{the } ((\text{the } (\kappa i))_{ts} j)) =$
 $sh (\text{the } ((\text{the } (\kappa i')_{ts} j')))) \wedge$$

($vs = []$)

end

5 Execution rules for threads

theory *KPL-execution-thread* **imports**

KPL-state

begin

Evaluate a local expression down to a word

fun *eval-word* :: *local-expr* \Rightarrow *thread-state* \Rightarrow *word*

where

eval-word (*Loc* (*Var* v)) $\tau = l \tau$ (*Inl* v)

| *eval-word* *Lid* $\tau = l \tau$ *LID*

| *eval-word* *Gid* $\tau = l \tau$ *GID*

| *eval-word* *eTrue* $\tau = 1$

| *eval-word* ($e1 \wedge e2$) $\tau =$
 (*eval-word* $e1 \tau * \text{eval-word } e2 \tau$)

| *eval-word* ($\neg e$) $\tau = (\text{if } \text{eval-word } e \tau = 0 \text{ then } 1 \text{ else } 0)$

Evaluate a local expression down to a boolean

fun *eval-bool* :: *local-expr* \Rightarrow *thread-state* \Rightarrow *bool*

where

eval-bool $e \tau = (\text{eval-word } e \tau \neq 0)$

Abstraction level: none, equality abstraction, or adversarial abstraction

datatype *abs-level* = *No-Abst* | *Eq-Abst* | *Adv-Abst*

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

inductive *step-t*

:: *abs-level* \Rightarrow (*thread-state* \times *pred-basic-stmt*) \Rightarrow *thread-state* \Rightarrow *bool*

where

T-Disabled:

$\neg (\text{eval-bool } p \tau) \Longrightarrow \text{step-t } a (\tau, (b, p)) \tau$

| *T-Assign*:

$\llbracket \text{eval-bool } p \tau ; l' = (l \tau) (\text{Inl } v := \text{eval-word } e \tau) \rrbracket$
 $\Longrightarrow \text{step-t } a (\tau, (\text{Assign } (\text{Var } v) e, p)) (\tau (| l := l' |))$

| *T-Read*:

$\llbracket \text{eval-bool } p \tau ; l' = (l \tau) (\text{Inl } v := \text{sh } \tau (\text{eval-word } e \tau)) ;$
 $R' = R \tau \cup \{ \text{eval-word } e \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} \rrbracket$
 $\Longrightarrow \text{step-t } a (\tau, (\text{Read } (\text{Var } v) e, p)) (\tau (| l := l', R := R' |))$

| *T-Write*:

$\llbracket \text{eval-bool } p \tau ;$
 $\text{sh}' = (\text{sh } \tau) (\text{eval-word } e1 \tau := \text{eval-word } e2 \tau) ;$
 $W' = W \tau \cup \{ \text{eval-word } e1 \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} \rrbracket$

$\Rightarrow \text{step-t a } (\tau, (\text{Write } e1 \ e2, p)) (\tau (| \text{sh} := \text{sh}', W := W' |))$
| *T-Read-Adv*:
 $\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) (\text{Inl } v := \text{asterisk}) ;$
 $R' = R \ \tau \cup \{ \text{eval-word } e \ \tau \} \rrbracket$
 $\Rightarrow \text{step-t Adv-Abst } (\tau, (\text{Read } (\text{Var } v) \ e, p)) (\tau (| l := l', R := R' |))$
| *T-Write-Adv*:
 $\llbracket \text{eval-bool } p \ \tau ; W' = W \ \tau \cup \{ \text{eval-word } e1 \ \tau \} \rrbracket$
 $\Rightarrow \text{step-t Adv-Abst } (\tau, (\text{Write } e1 \ e2, p)) (\tau (| \text{sh} := \text{sh}', W := W' |))$

Rephrasing *T-Assign* to make it more usable

lemma *T-Assign-helper*:

$\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) (\text{Inl } v := \text{eval-word } e \ \tau) ; \tau' = \tau (| l := l' |) \rrbracket$
 $\Rightarrow \text{step-t a } (\tau, (\text{Assign } (\text{Var } v) \ e, p)) \tau'$

by (*auto simp add: step-t.T-Assign*)

Rephrasing *T-Read* to make it more usable

lemma *T-Read-helper*:

$\llbracket \text{eval-bool } p \ \tau ; l' = (l \ \tau) (\text{Inl } v := \text{sh } \tau (\text{eval-word } e \ \tau)) ;$
 $R' = R \ \tau \cup \{ \text{eval-word } e \ \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} ;$
 $\tau' = \tau (| l := l', R := R' |) \rrbracket$
 $\Rightarrow \text{step-t a } (\tau, (\text{Read } (\text{Var } v) \ e, p)) \tau'$

by (*auto simp add: step-t.T-Read*)

Rephrasing *T-Write* to make it more usable

lemma *T-Write-helper*:

$\llbracket \text{eval-bool } p \ \tau ;$
 $\text{sh}' = (\text{sh } \tau) (\text{eval-word } e1 \ \tau := \text{eval-word } e2 \ \tau) ;$
 $W' = W \ \tau \cup \{ \text{eval-word } e1 \ \tau \} ; a \in \{ \text{No-Abst}, \text{Eq-Abst} \} ;$
 $\tau' = \tau (| \text{sh} := \text{sh}', W := W' |) \rrbracket$
 $\Rightarrow \text{step-t a } (\tau, (\text{Write } e1 \ e2, p)) \tau'$

by (*auto simp add: step-t.T-Write*)

end

6 Execution rules for groups

theory *KPL-execution-group imports*

KPL-execution-thread

begin

Intra-group race detection

definition *group-race*

$:: \text{lid set} \Rightarrow (\text{lid} \rightarrow \text{thread-state}) \Rightarrow \text{bool}$

where *group-race* $T \ \gamma \equiv$

$\exists j \in T. \exists k \in T. j \neq k \wedge$

$W (\text{the } (\gamma \ j)) \cap (R (\text{the } (\gamma \ k)) \cup W (\text{the } (\gamma \ k))) \neq \{ \}$

The constraints for the *merge* map

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

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

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

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

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

lemma *merge-is-unique*:
assumes $\text{dom } \gamma = T$
assumes $\neg (\text{group-race } T \gamma)$
shows $\exists! z'. \text{pre-merge } T \gamma z z'$
apply (*insert assms*)
apply (*drule no-races-imp-no-write-overlap*)
apply (*intro allI ex-ex1I*)
apply (*metis pre-merge.intros*)
apply *clarify*
proof –
fix $z1 z2$
assume $a: \forall i \in \text{dom } \gamma. \forall j \in \text{dom } \gamma. i \neq j \longrightarrow W (\text{the } (\gamma i)) \cap W (\text{the } (\gamma j)) = \{\}$
 $\{\}$
assume $\text{pre-merge } (\text{dom } \gamma) \gamma z z1$
and $\text{pre-merge } (\text{dom } \gamma) \gamma z z2$
thus $z1 = z2$
apply (*elim pre-merge-inv*)
apply (*rename-tac j1 j2*)
apply (*case-tac j1 = j2*)
apply *auto[1]*
apply *simp*
apply (*subgoal-tac* $W (\text{the } (\gamma j1)) \cap W (\text{the } (\gamma j2)) = \{\}$)
apply *auto[1]*
apply (*auto simp add: a*)
done
qed

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*

$\text{:: } \text{abs-level} \Rightarrow \text{gid} \Rightarrow (\text{gid} \rightarrow \text{lid set}) \Rightarrow (\text{group-state} \times \text{pred-stmt}) \Rightarrow \text{group-state option} \Rightarrow \text{bool}$

where

G-Race:

$\llbracket \forall j \in \text{the } (T \ i). \text{ step-t } a \ (\text{the } (\gamma_{ts} \ j), (s, p)) \ (\text{the } (\gamma'_{ts} \ j)) ;$
 $\text{group-race } (\text{the } (T \ i)) \ ((\gamma' \text{:: } \text{group-state})_{ts}) \rrbracket$
 $\Rightarrow \text{step-g } a \ i \ T \ (\gamma, (\text{Basic } s, p)) \ \text{None}$

| *G-Basic:*

$\llbracket \forall j \in \text{the } (T \ i). \text{ step-t } a \ (\text{the } (\gamma_{ts} \ j), (s, p)) \ (\text{the } (\gamma'_{ts} \ j)) ;$
 $\neg (\text{group-race } (\text{the } (T \ i)) \ (\gamma'_{ts})) ;$
 $R\text{-group } \gamma' = R\text{-group } \gamma \cup (\bigcup j \in \text{the } (T \ i). (\{j\} \times R \ (\text{the } (\gamma'_{ts} \ j)))) ;$
 $W\text{-group } \gamma' = W\text{-group } \gamma \cup (\bigcup j \in \text{the } (T \ i). (\{j\} \times W \ (\text{the } (\gamma'_{ts} \ j)))) \rrbracket$
 $\Rightarrow \text{step-g } a \ i \ T \ (\gamma, (\text{Basic } s, p)) \ (\text{Some } \gamma')$

| *G-No-Op:*

$\forall j \in \text{the } (T \ i). \neg (\text{eval-bool } p \ (\text{the } (\gamma_{ts} \ j)))$
 $\Rightarrow \text{step-g } a \ i \ T \ (\gamma, (\text{Barrier}, p)) \ (\text{Some } \gamma)$

| *G-Divergence:*

$\llbracket j \neq k ; j \in \text{the } (T \ i) ; k \in \text{the } (T \ i) ;$
 $\text{eval-bool } p \ (\text{the } (\gamma_{ts} \ j)) ; \neg (\text{eval-bool } p \ (\text{the } (\gamma_{ts} \ k))) \rrbracket$
 $\Rightarrow \text{step-g } a \ i \ T \ (\gamma, (\text{Barrier}, p)) \ \text{None}$

| *G-Sync:*

$\llbracket \forall j \in \text{the } (T \ i). \text{ eval-bool } p \ (\text{the } (\gamma_{ts} \ j)) ;$
 $\forall j \in \text{the } (T \ i). \text{ the } (\gamma'_{ts} \ j) = (\text{the } (\gamma_{ts} \ j)) \ (|$
 $\text{sh} := \text{merge } P \ (\gamma_{ts}), R := \{\}, W := \{\} \ |) \rrbracket$
 $\Rightarrow \text{step-g } \text{No-Abst } i \ T \ (\gamma, (\text{Barrier}, p)) \ (\text{Some } \gamma')$

| *G-Sync-Eq:*

$\llbracket \forall j \in \text{the } (T \ i). \text{ eval-bool } p \ (\text{the } (\gamma_{ts} \ j)) ;$
 $\forall j \in \text{the } (T \ i). \text{ the } (\gamma'_{ts} \ j) = (\text{the } (\gamma_{ts} \ j)) \ (|$
 $\text{sh} := \text{sh}', R := \{\}, W := \{\} \ |) \rrbracket$
 $\Rightarrow \text{step-g } \text{Eq-Abst } i \ T \ (\gamma, (\text{Barrier}, p)) \ (\text{Some } \gamma')$

| *G-Sync-Adv:*

$\llbracket \forall j \in \text{the } (T \ i). \text{ eval-bool } p \ (\text{the } (\gamma_{ts} \ j)) ;$
 $\forall j \in \text{the } (T \ i). \exists \text{sh}'. \text{ the } (\gamma'_{ts} \ j) = (\text{the } (\gamma_{ts} \ j)) \ (|$
 $\text{sh} := \text{sh}', R := \{\}, W := \{\} \ |) \rrbracket$
 $\Rightarrow \text{step-g } \text{Adv-Abst } i \ T \ (\gamma, (\text{Barrier}, p)) \ (\text{Some } \gamma')$

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

lemma *G-No-Op-helper:*

$\llbracket \forall j \in \text{the } (T \ i). \neg (\text{eval-bool } p \ (\text{the } (\gamma_{ts} \ j))) ; \gamma = \gamma' \rrbracket$
 $\Rightarrow \text{step-g } a \ i \ T \ (\gamma, (\text{Barrier}, p)) \ (\text{Some } \gamma')$

by (*simp add: step-g.G-No-Op*)

end

7 Execution rules for kernels

theory *KPL-execution-kernel* **imports**

KPL-execution-group

begin

Inter-group race detection

definition *kernel-race*

$:: \text{gid set} \Rightarrow (\text{gid} \rightarrow \text{group-state}) \Rightarrow \text{bool}$

where *kernel-race* $G \kappa \equiv$

$\exists i \in G. \exists j \in G. i \neq j \wedge$
 $(\text{snd } ' (W\text{-group } (\text{the } (\kappa i)))) \cap$
 $(\text{snd } ' (R\text{-group } (\text{the } (\kappa j)))) \cup \text{snd } ' (W\text{-group } (\text{the } (\kappa j))) \neq \{\}$

Replaces top-level *Break* with $v := \text{true}$

fun *belim* $:: \text{stmt} \Rightarrow V \Rightarrow \text{stmt}$

where

belim (*Basic* b) $v = \text{Basic } b$
 $| \text{belim } (S1 ;; S2) v = (\text{belim } S1 v ;; \text{belim } S2 v)$
 $| \text{belim } (\text{Local } n S) v = \text{Local } n (\text{belim } S v)$
 $| \text{belim } (\text{If } e S1 S2) v = \text{If } e (\text{belim } S1 v) (\text{belim } S2 v)$
 $| \text{belim } (\text{While } e S) v = \text{While } e S$

 $| \text{belim } (\text{Call } f e) v = \text{Call } f e$
 $| \text{belim } \text{Barrier } v = \text{Barrier}$
 $| \text{belim } \text{Break } v = \text{Basic } (\text{Assign } (\text{Var } v) e\text{True})$
 $| \text{belim } \text{Continue } v = \text{Continue}$
 $| \text{belim } \text{Return } v = \text{Return}$

Replaces top-level *Continue* with $v := \text{true}$

fun *celim* $:: \text{stmt} \Rightarrow V \Rightarrow \text{stmt}$

where

celim (*Basic* b) $v = \text{Basic } b$
 $| \text{celim } (S1 ;; S2) v = (\text{celim } S1 v ;; \text{celim } S2 v)$
 $| \text{celim } (\text{Local } n S) v = \text{Local } n (\text{celim } S v)$
 $| \text{celim } (\text{If } e S1 S2) v = \text{If } e (\text{celim } S1 v) (\text{celim } S2 v)$
 $| \text{celim } (\text{While } e S) v = \text{While } e S$

 $| \text{celim } (\text{Call } f e) v = \text{Call } f e$
 $| \text{celim } \text{Barrier } v = \text{Barrier}$
 $| \text{celim } \text{Break } v = \text{Break}$
 $| \text{celim } \text{Continue } v = \text{Basic } (\text{Assign } (\text{Var } v) e\text{True})$
 $| \text{celim } \text{Return } v = \text{Return}$

subst-basic-stmt $n v \text{ loc}$ replaces n with v inside loc

fun *subst-loc* $:: \text{name} \Rightarrow V \Rightarrow \text{loc} \Rightarrow \text{loc}$

where

subst-loc $n v (\text{Var } w) = \text{Var } w$
 $| \text{subst-loc } n v (\text{Name } m) = (\text{if } n = m \text{ then } \text{Var } v \text{ else } \text{Name } m)$

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

fun *subst-local-expr*

:: name \Rightarrow *V* \Rightarrow *local-expr* \Rightarrow *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 \wedge^* e2) =*
(subst-local-expr n v e1 \wedge^ subst-local-expr n v e2)*
| *subst-local-expr n v (\neg^* e) = \neg^* (subst-local-expr n v e)*

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

fun *subst-basic-stmt* *:: name* \Rightarrow *V* \Rightarrow *basic-stmt* \Rightarrow *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* \Rightarrow *V* \Rightarrow *stmt* \Rightarrow *stmt* \Rightarrow *bool*

where

subst-stmt n v (Basic b) (Basic (subst-basic-stmt n v b))
| $\llbracket \text{subst-stmt } n \ v \ S1 \ S1' ; \text{subst-stmt } n \ v \ S2 \ S2' \rrbracket \Longrightarrow$
subst-stmt n v (S1 ;; S2) (S1' ;; S2')
| $\llbracket m \neq n ; \text{subst-stmt } n \ v \ S \ S' \rrbracket \Longrightarrow$
subst-stmt n v (Local m S) (Local m S')
| $\llbracket \text{subst-stmt } n \ v \ S1 \ S1' ; \text{subst-stmt } n \ v \ S2 \ S2' \rrbracket \Longrightarrow$
subst-stmt n v (If e S1 S2) (If e S1' S2')
| *subst-stmt n v S S' \Longrightarrow 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* \Rightarrow *proc-name* \Rightarrow *V* \Rightarrow *stmt*

where *param-subst fs f u* \equiv

let proc = THE proc. proc \in set fs \wedge proc-name proc = f in
THE S'. subst-stmt (param proc) u (body proc) S'

Replace *Return* with *v := true*

fun *relin* *:: stmt* \Rightarrow *V* \Rightarrow *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 $fresh :: V \Rightarrow V\ list \Rightarrow bool$

where $fresh\ v\ vs \equiv v \notin set\ vs$

The rules of Figure 6

inductive $step-k$

$:: abs-level \Rightarrow proc\ list \Rightarrow threadset \Rightarrow kernel-state \Rightarrow kernel-state\ option \Rightarrow bool$

where

K-Inter-Group-Race:

$\llbracket \forall i \in G. step-g\ a\ i\ T\ (the\ (\kappa\ i),\ (Basic\ b,\ p))\ (Some\ (the\ (\kappa'\ i))) \rrbracket ;$
 $kernel-race\ P\ \kappa' \rrbracket \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Basic\ b,\ p)\ \# ss,\ vs)\ None$

| *K-Intra-Group-Race:*

$\llbracket i \in G; step-g\ a\ i\ T\ (the\ (\kappa\ i),\ (Basic\ s,\ p))\ None \rrbracket \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Basic\ s,\ p)\ \# ss,\ vs)\ None$

| *K-Basic:*

$\llbracket \forall i \in G. step-g\ a\ i\ T\ (the\ (\kappa\ i),\ (Basic\ b,\ p))\ (Some\ (the\ (\kappa'\ i))) \rrbracket ;$
 $\neg (kernel-race\ G\ \kappa') \rrbracket \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Basic\ b,\ p)\ \# ss,\ vs)\ (Some\ (\kappa',ss,\ vs))$

| *K-Divergence:*

$\llbracket i \in G; step-g\ a\ i\ T\ (the\ (\kappa\ i),\ (Barrier,\ p))\ None \rrbracket \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Barrier,\ p)\ \# ss,\ vs)\ None$

| *K-Sync:*

$\llbracket \forall i \in G. step-g\ a\ i\ T\ (the\ (\kappa\ i),\ (Barrier,\ p))\ (Some\ (the\ (\kappa'\ i))) \rrbracket ;$
 $\neg (kernel-race\ G\ \kappa') \rrbracket \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Barrier,\ p)\ \# ss,\ vs)\ (Some\ (\kappa',ss,\ vs))$

| *K-Seq:*

$step-k\ a\ fs\ (G,T)\ (\kappa,\ (S1\ ;;\ S2,\ p)\ \# ss,\ vs)$
 $(Some\ (\kappa,\ (S1,\ p)\ \# (S2,\ p)\ \# ss,\ vs))$

| *K-Var:*

$fresh\ v\ vs \implies$
 $step-k\ a\ fs\ (G,T)\ (\kappa,\ (Local\ n\ S,\ p)\ \# ss,\ vs)$
 $(Some\ (\kappa,\ (THE\ S'.\ subst-stmt\ n\ v\ S\ S',\ p)\ \# ss,\ v\ \# vs))$

| *K-If:*

$fresh\ v\ vs \implies$

```

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) (κ, (WhileDyn e S, p) # ss, vs) (Some (κ, ss, vs))
| K-Call:
[[ fresh u vs ; fresh v vs ; u ≠ v ; s = param-subst fs f u ]] ⇒
step-k a fs (G,T) (κ, (Call f e, p) # ss, vs )
(Some (κ, (Basic (Assign (Var u) e) ;; relim s v,
p ∧* ¬* (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.