Journal of Information Processing Vol.0 No.0 1–10 (??? 1992)

[DOI: 10.2197/ipsjjip.0.1]

Regular Paper

Proof of Soundness of Concurrent Separation Logic for GPGPU in Coq

Izumi Asakura[1,a)] Hidehiko Masuhara[1,b)] Tomoyuki Aotani[1,c)]

Received: May 8, 2015, Accepted: July 28, 2015

Abstract: We design a concurrent separation logic for GPGPU, namely GPUCSL, and prove its soundness by using Coq. GPUCSL is based on a CSL proposed by Blom et al., which is for automatic verification of GPGPU kernels, but employs different inference rules because the rules in Blom's CSL are not standard. For example, Blom's CSL does not have a frame rule. Our CSL is a simple extension of the original CSL, and it is more suitable as a basis of advanced properties proposed for other studies on CSLs. Our soundness proof is based on Vafeiadis' method, which is for a CSL with a fork-join concurrency model. The proof reveals two problems in Blom's approach in terms of soundness and extensibility. First, their assumption that thread ID independence of a kernel implies barrier divergence freedom does not hold. Second, it is not easy to extend their proof to other CSLs with a frame rule. Although our CSL covers only a subset of CUDA, our preliminary experiment shows that it is useful and expressive enough to verify a simple kernel with barriers.

Keywords: Concurrent separation logic, Coq, GPGPU, barrier divergence

1. Introduction

GPGPU (general-purpose computing on graphics processing units) is a method that allows GPUs to be used for general purpose computation. It is used in many applications because GPUs provide high parallelism at a low price.

However, the programmers need to write GPGPU programs carefully in order not to cause data races (e.g., by using synchronization among threads), because the programs run in an SPMT (single program multiple threads) manner.

One of the pitfalls GPGPU programs is *barrier divergence* [9]. CUDA, which is one of the widely used programming languages for GPGPU, has a synchronization instruction that succeeds when and only when all threads reach the barrier instruction at the same location in a program. If some threads do not execute any barrier instructions or reach a barrier instruction at a different location in the program, the behavior of the program is unspecified.

Blom et al. proposed an extension of concurrent separation logic (CSL) [10] for verifying GPGPU programs [3]. They extended the CSL for SPMT programs and made an automated verifier. The inference rules of Blom's CSL differ from the standard CSL in the following points. (i) Their CSL does not have a frame rule. (ii) In their CSL, assertions must be separated into those on resources and those on functions. They also proposed *thread ID independence* as a sufficient condition for freedom from barrier divergence. However, they did not prove the soundness of the condition.

Here, we designed a CSL for GPGPU, namely GPUCSL, on the basis of Blom's CSL and Vafeiadis' CSL [13] and proved its soundness by using the Coq proof assistant [11]. The inference rules of GPUCSL is designed to be similar to standard CSL. To prove soundness, we applied Vafeiadis' soundness proof for a CSL based on a fork-join concurrency model [13]. We also formalized thread ID independence as a type system and proved its soundness.

In proving the soundness, we identified the following problems in Blom's proof. First, it is hard to apply the proof strategy to CSLs with the frame rule. This is because one of the lemmas in their proof does not hold in CSLs with the frame rule. Second, Blom et al. overlooked the prerequisite to the thread ID independence of a program, namely the absence of data races in the program.

The rest of this paper is organized as follows. We present our object language in Sec. 2 and the inference rules and definition of soundness of GPUCSL in Sec. 3. We give a sketch of the soundness proof of GPUCSL in Sec. 4 and discuss differences between Blom's CSL and GPUCSL in Sec. 5. We show an example of verifying a simple program in Sec. 6 and discuss related work in Sec. 7. Finally, we present our conclusion and mention future work in Sec. 8.

2. Language

The object language of GPUCSL is a simplified version of CUDA C [9]. In CUDA C, GPUs and CPUs have their own memories and carry out computations in the following steps. (1) A program running on a CPU (*the host program*) allocates memory areas that are used by a program running

[1] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology
[a)] asakura.i.aa@m.titech.ac.jp
[b)] masuhara@acm.org
[c)] aotani@is.titech.ac.jp

$$E ::= x \mid n \mid E_1 + E_2 \mid \ldots$$
$$B ::= E_1 = E_2 \mid E_1 \leq E_2 \mid !B \mid B_1 \&\& B_2 \mid \ldots$$
$$C ::= \textbf{skip} \mid x := E \mid x := [E] \mid [E_1] := E_2 \mid C_1; C_2 \mid \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{while } B \textbf{ do } C \mid \textbf{barrier}_b$$

Fig. 1 Syntax of the WhileB language

$$\text{G-Step}\frac{(C_i, s_i, h) \to_t (C', s', h')}{(\overline{C}, \overline{s}, h) \to_g (\overline{C}[i := C'], \overline{s}[i := s'], h')} \qquad \text{G-Barrier}\frac{\exists b, \forall i \in Tid, \mathsf{wait}(C_i) = (b, C_i')}{(\overline{C}, \overline{s}, h) \to_g (\overline{C'}, \overline{s}, h)}$$
$$\text{G-Abort}\frac{\exists i \in Tid, (C_i, s_i, h) \to_t \textbf{abort}}{(\overline{C}, \overline{s}, h) \to_g \textbf{abort}}$$

Fig. 3 Operational semantics for parallel execution

```
1 x := [a + tid];
2 barrier0;
3 t = tid + 1;
4 if (tid == T - 1) {
5   t = 0
6 }
7 [a + t] := x;
```

Fig. 2 Program example: rotate

$$\begin{aligned}\mathsf{wait}(\textbf{barrier}_b) &= (b, \textbf{skip})\\ \mathsf{wait}(C_1; C_2) &= (b', C_1'; C_2) \quad (\mathsf{wait}(C_1) = (b', C_1'))\\ \mathsf{wait}(C) &= \bot \quad \text{(otherwise)}\end{aligned}$$

Fig. 4 Definition of the wait function

on a GPU (*the kernel*). (2) The host program invokes the kernel with the number of threads and the addresses of the memory areas allocated in step (1). (3) The GPU spawns the specified number of threads in step (2), each of which starts executing the kernel from its first instruction. Each thread uses its own registers and the global memory, which is shared with all the other threads (i.e., based on shared memory parallelism). One of the registers holds the thread ID. The model for synchronization among threads is based on barriers.

With barrier synchronization, a barrier instruction only succeeds when all threads reach the barrier instruction at the same location in a kernel. A state when two threads reach two different barriers is called *barrier divergence* [9], and its behavior is unspecified. One of the properties that GPUCSL verifies is that kernels do not cause barrier divergence.

In this paper, we only discuss execution of a kernel on a GPU. We define a language for writing a kernel by extending the "while" language with instructions for barrier synchronization and for reading and writing to the global memory (*WhileB* language). We discuss the features in CUDA C that are omitted in the WhileB language in Sec. 8.

2.1 Syntax rule

Fig. 1 shows the syntax rules of WhileB. The meta-variables E, B and C denote arithmetic expressions, Boolean expressions, and commands, respectively. x and n denote local variables and integer values, respectively. A local variable corresponds to a register that every thread has. Note that E and B have no instructions that read or write to the global memory. $x := [E_1]$ and $[E_1] := E_2$ are respectively read and write instructions for the global memory. $\textbf{barrier}_b$ is the barrier instruction. The instruction is indexed by b to account for barrier divergence. Each barrier instruction in a program must have an index b that is different from the index of any other barrier in the program.

Fig. 2 shows an example program rotate in WhileB, where T is the number of threads as well as the length of array `a`. Each thread rotates an element of the array `a` to the right by one offset. In other words, an element at index i is moved to the index $((i+1) \bmod T)$ when rotate finishes. Line 1 of rotate reads an element of `a` at index `tid`. The variable `tid` has a thread ID. Line 2 is a barrier instruction, which lets all the threads execute the write instruction after all threads have finished the read instruction. Lines 3–6 calculate the destination index `t`. Line 7 writes to `a` at index `t`.

2.2 Semantics

We define the semantics of WhileB by extending the semantics of Vafeiadis [13] with transition rules for barrier instructions. The semantics consist of rules for parallel execution $\to_g$ and rules for sequential execution $\to_t$. A state of the GPU is represented by a triple $(\overline{C}, \overline{s}, h)$, where $\overline{x}$ is an abbreviation of $x_0, x_1, \ldots, x_{T-1}$, and T is the number of threads. The elements of the triple denote the following states.

- $\overline{C}$ is a sequence of commands, where C_i is the command to be executed by the thread i.
- $\overline{s}$ is a sequence of variable environments (stacks), where s_i is the stack of thread i.
- h is the global memory.

A stack s_i is a function from variables Var to values Val ($s_i \in \mathsf{stack} = \mathsf{Var} \to \mathsf{Val}$.) A global memory h is a partial function from addresses Loc to values Val ($h \in \mathsf{heap} = \mathsf{Loc} \to \mathsf{Val} \cup \{\bot\}$.) Loc is the set of integers.

Fig. 3 defines the semantics of parallel execution ($\to_g$). The definition is represented in the form $(\overline{C}, \overline{s}, h) \to_g (\overline{C'}, \overline{s'}, h')$. The parallel execution denotes a GPU state transition from $(\overline{C}, \overline{s}, h)$ to $(\overline{C'}, \overline{s'}, h')$ by one-step execution of the GPU. The G-Step rule moves one of the threads by using the sequential execution $\to_t$. The G-Barrier rule moves all threads by one step when all threads reach a bar-

$$\text{T-Seq1}\ \frac{}{(\mathbf{skip}; C, s, h) \to_t (C, s, h)} \qquad \text{T-Seq2}\ \frac{(C_1, s, h) \to_t (C_1', s', h')}{(C_1; C_2, s, h) \to_t (C_1'; C_2, s', h')}$$

$$\text{T-If1}\ \frac{s(B) = \mathbf{true}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s, h) \to_t (C_1, s, h)} \qquad \text{T-If2}\ \frac{s(B) = \mathbf{false}}{(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s, h) \to_t (C_2, s, h)}$$

$$\text{T-While}\ \frac{}{(\mathbf{while}\ B\ \mathbf{do}\ C, s, h) \to_t (\mathbf{if}\ B\ \mathbf{then}\ (C; \mathbf{while}\ B\ \mathbf{do}\ C)\ \mathbf{else}\ \mathbf{skip}, s, h)}$$

$$\text{T-Assign}\ \frac{s(E) = v}{(x := E, s, h) \to_t (\mathbf{skip}, s[x := v], h)}$$

$$\text{T-Read}\ \frac{s(E) = v \quad h(v) = [v']}{(x := [E], s, h) \to_t (\mathbf{skip}, s[x := v'], h)} \qquad \text{T-Write}\ \frac{s(E_1) = v_1 \quad s(E_2) = v_2}{([E_1] := E_2, s, h) \to_t (\mathbf{skip}, s, h[v_1 := v_2])}$$

$$\text{T-ReadA}\ \frac{s(E) = v \quad h(v) = \bot}{(x := [E], s, h) \to_t \mathbf{abort}} \qquad \text{T-WriteA}\ \frac{s(E_1) = v \quad h(v) = \bot}{([E_1] := E_2, s, h) \to_t \mathbf{abort}}$$

Fig. 5 Operational semantics for sequential execution

$$\begin{aligned} s(n) &= n \\ s(E_1 + E_2) &= s(E_1) + s(E_2) \\ s(E_1 = E_2) &= s(E_1) = s(E_2) \\ s(E_1 \le E_2) &= s(E_1) \le s(E_2) \\ s(!B) &= \neg s(B) \\ s(B_1 \&\& B_2) &= s(B_1) \wedge s(B_2) \end{aligned}$$

Fig. 6 Definition of an extended stack

$$\text{Parallel}\ \frac{\begin{array}{c} P \Rightarrow \bigstar_i P_i \\ \bigstar_i Q_i \Rightarrow Q \\ \forall i \in \mathit{Tid},\ BS, i \vdash_{seq} \{P_i \wedge \mathtt{tid} = i\}\ C\ \{Q_i\} \\ \Gamma \vdash C : \tau \\ \forall b,\ \bigstar_{i \in \mathit{Tid}} BS(i,b)_{pre} \Rightarrow \bigstar_{i \in \mathit{Tid}} BS(i,b)_{post} \\ \forall i \in \mathit{Tid}\ b, BS(i,b)_{pre/post} \text{ is precise} \\ \forall i \in \mathit{Tid} \text{ and } b, P_i, Q_i \text{ and } BS(i,b) \text{ are tid independent} \end{array}}{\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}}$$

Fig. 7 Proof rules for parallel execution

rier instruction. This rule was newly added by the authors. Tid is the set of thread ID, namely $\mathit{Tid} = \{0, 1, \ldots, T-1\}$. The relation $\mathsf{wait}(C) = (B, C')$ denotes that all the threads in the kernel C reach a barrier with index b, and the next set of commands is C' (Fig. 4). The G-Abort rule aborts the kernel if any of the threads aborts.

Fig. 5 defines the sequential execution ($\to_t$). $\to_t$ are the same as those in the standard While language. Note that the domain of a stack s is extended to arithmetic and Boolean expressions, as shown in Fig. 6.

3. GPUCSL

CSL verifies that a concurrent program C satisfies a specification $\{P\}\ C\ \{Q\}$. The specification reads: when C starts with a state satisfying P, (i) C does not abort during the execution and (ii) the memory state when C terminates satisfies Q. P and Q are predicates on a pair of a stack and a heap, namely *assertion*.

GPUCSL verifies that a kernel meets a specification. Its inference rules are based on those for Blom's CSL and Vafeiadis' CSL. GPUCSL ensures barrier divergence freedom in addition to the above properties (i) and (ii). The inference rules include the Parallel rule (Fig. 7) to verify parallel execution of a kernel. The premise of the rule check the following properties of the kernel.

- L1 The precondition implies a separating conjunction of preconditions of threads.
- L2 There are postconditions of threads whose separating conjunction implies the postcondition of the specification.
- L3 For all threads, the specification $\{P_i \wedge \mathtt{tid} = i\}\ C\ \{Q_i\}$ is satisfied under the sequential execution semantics.
- L4 The kernel is thread ID independent. Intuitively, this means that barrier instructions in the kernel appear only in execution contexts that do not depend on the thread ID.
- L5 In every barrier specification BS, the memory resources redistributed before and after a barrier synchronization instruction are exactly the same. This means that the threads correctly exchange the memory areas used before and after the barrier synchronization instruction.

The conditions at line 6 and line 7 are not important and will be explained later.

Note that GPUCSL can only verify programs without data races, as it is based on separation logic [10].

3.1 Inference rules

GPUCSL is a fraction-based permission CSL (FPCSL) [4]. Assertions of FPCSL are predicates on a pair of a stack and a permission-heap (pheap). pheap is a function of type $\mathsf{Loc} \to (\mathsf{Perm} \times \mathsf{Val}) \cup \{\bot\}$, where Perm is a set of rational numbers that are greater than 0 and less than or equal to 1. When a thread has a pheap h, and h satisfies $h(l) = (1, v)$ for some address l and some value v, the thread has permission to read from and write to address l. If h satisfies $h(l) = (\pi, v)$ for some $\pi < 1$, the thread only has read permission.

Now let us define the sum of two pheaps ($\uplus$). First, we define a binary operator ($\oplus$) over $\mathsf{Perm} \times \mathsf{Val}$.

$$(\pi_1, v_1) \oplus (\pi_2, v_2) = \begin{cases} (\pi_1 + \pi_2, v_1) & (v_1 = v_2 \wedge \pi_1 + \pi_2 \le 1) \\ \bot & (\text{otherwise}) \end{cases}$$

For all pheaps h_1 and h_2, $h_1 \bot h_2$ denotes that, for all $l \in \mathbf{dom}(h_1) \cap \mathbf{dom}(h_2)$, $h_1(l) \oplus h_2(l) \neq \bot$, where $\mathbf{dom}(h)$ is $\{l \mid h(l) \neq \bot\}$. $h_1 \uplus h_2$ is defined if and only if $h_1 \bot h_2$:

$$(h_1 \uplus h_2)(l) =$$

$$\text{Skip}\ \frac{}{BS, i \vdash_{seq} \{Q\}\ \textbf{skip}\ \{Q\}}$$

$$\text{Seq}\ \frac{BS, i \vdash_{seq} \{P\}\ C_1\ \{Q\} \quad BS, i \vdash_{seq} \{Q\}\ C_2\ \{R\}}{BS, i \vdash_{seq} \{P\}\ C_1; C_2\ \{R\}}$$

$$\text{While}\ \frac{BS, i \vdash_{seq} \{P \wedge B\}\ C\ \{P\}}{BS, i \vdash_{seq} \{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P \wedge \neg B\}}$$

$$\text{Assign}\ \frac{}{BS, i \vdash_{seq} \{P[E/x]\}\ x := E\ \{P\}}$$

$$\text{Write}\ \frac{}{BS, i \vdash_{seq} \{E_1 \mapsto^1 E_0\}\ [E_1] := E_2\ \{E_1 \mapsto^1 E_2\}}$$

$$\text{Frame}\ \frac{BS, i \vdash_{seq} \{P\}\ C\ \{Q\} \quad \text{fv}(R) \cap \text{wr}(C) = \emptyset}{BS, i \vdash_{seq} \{P \star R\}\ C\ \{Q \star R\}}$$

$$\text{If}\ \frac{BS, i \vdash_{seq} \{P \wedge B\}\ C_1\ \{Q\} \quad BS, i \vdash_{seq} \{P \wedge \neg B\}\ C_2\ \{Q\}}{BS, i \vdash_{seq} \{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}$$

$$\text{Barrier}\ \frac{}{BS, i \vdash_{seq} \{BS(i,b)_{pre}\}\ \textbf{barrier}_b\ \{BS(i,b)_{post}\}}$$

$$\text{READ}\ \frac{x \notin \text{fv}(E_1) \quad x \notin \text{fv}(E_2)}{BS, i \vdash_{seq} \{E_1 \mapsto^\pi E_2\}\ x := [E_1]\ \{E_1 \mapsto^\pi E_2 \wedge x = E_2\}}$$

$$\text{Conseq}\ \frac{P \Rightarrow P' \quad BS, i \vdash_{seq} \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{BS, i \vdash_{seq} \{P\}\ C\ \{Q\}}$$

Fig. 8 Proof rules for sequential execution

$$\begin{cases} h_1(l) \oplus h_2(l) & (h_1(l) \neq \bot \wedge h_2(l) \neq \bot) \\ h_1(l) & (h_1(l) \neq \bot \wedge h_2(l) = \bot) \\ h_2(l) & (h_1(l) = \bot \wedge h_2(l) \neq \bot) \\ \bot & (\text{otherwise}). \end{cases}$$

GPUCSL has, in addition to logical formulas, the following logical operators: **emp**, $E_1 \mapsto^\pi E_2$ and $P \star Q$. The semantics of these operators are as follows.

$$\begin{aligned} s, h \models \textbf{emp} &\iff \forall l, h(l) = \bot \\ s, h \models E_1 \mapsto^\pi E_2 &\iff h(s(E_1)) = (\pi, s(E_2)) \wedge \\ &\quad\ \forall l \neq s(E_1), h(l) = \bot \\ s, h \models P \star Q &\iff \exists h_1\ h_2, h = h_1 \uplus h_2 \wedge \\ &\quad\ (s, h_1 \models P) \wedge (s, h_2 \models Q) \end{aligned}$$

We also use the following notations: $\biguplus_{i \in Tid} h_i = h_0 \uplus h_1 \uplus \ldots \uplus h_{T-1}$, and $\bigstar_{i \in Tid} P_i = P_0 \star P_1 \star \ldots \star P_{T-1}$.

In order to extend assertions to predicates on a pair of a stack and a heap, we convert heaps into pheaps by using an auxiliary function to_pheap.

$$\text{to\_pheap}(h)(l) = \begin{cases} (1, v) & (h(l) = v \neq \bot) \\ \bot & (h(l) = \bot) \end{cases}$$

Conversely, we can regard a pheap h as a heap if h satisfies $\forall l \in \textbf{dom}(h), \exists v, h(l) = (1, v) \vee h(l) = \bot$. We write this condition as $\text{hdef}(h)$.

Figures 7 and 8 show the inference rules of GPUCSL. Intuitively, $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}$ reads: under a typing environment Γ and a barrier specification BS, a specification $\{P\}\ C\ \{Q\}$ can be proved. GPUCSL has rules for barrier synchronization (the barrier rule) and parallel execution (the parallel rule) in addition to the rules for the separation logic by Vafeiadis [13].

The frame rule extends a proven specification $\{P\}\ C\ \{Q\}$ by adding a resource R that is not referenced in C. The sets $\text{fv}(R)$ and $\text{wr}(C)$ appearing in the premise respectively denote the set of variables appearing in R and the set of variables appearing in C on the left hand side of assignment statements.

The barrier rule redistributes the resources to threads according to the barrier specification. A barrier specification BS specifies the resources $BS(i,b)_{pre}$ and $BS(i,b)_{post}$, which denote the resource returned by thread i upon arrival of the barrier b and the resource allocated to thread i after the barrier synchronization, respectively. In order to guarantee proper exchange of resources before and after the barrier synchronization, BS must satisfy the condition that the separating conjunction of the returned resources implies the separating conjunction of the allocated resources, i.e., $\bigstar_{i \in Tid} BS(i,b)_{pre} \Rightarrow \bigstar_{i \in Tid} BS(i,b)_{post}$. This condition does not appear in the premise of the barrier rule, but does appears in the premise of the parallel rule.

The parallel rule verifies that a kernel meets a specification. Lines 1 and 2 in the premise represent distribution/aggregation of resources to/from each thread. Line 3 verifies each thread, and $\texttt{tid} = i$ in the precondition initializes the program variable tid to i. Line 4 means that the command is thread ID independent (Sec. 3.2). Line 5 means that each barrier specification properly redistributes resources. Line 6 represents that each barrier specification is precise, where assertion P is precise if and only if for all pheaps h, there exists at most one sub-pheap of h which satisfies P. Line 7 means that all preconditions, postconditions, and the barrier specifications are thread ID independent. An assertion P is thread ID independent if and only if all variables in P are thread ID independent. Any thread ID independent variable has the same value in all threads when a kernel terminates or performs barrier synchronization. The condition requires that the specifications are written by only using such variables.

3.2 Thread ID independence

Blom et al. proposed thread ID independence as a condition that guarantees barrier divergence freedom of kernels [3]. An instruction or a variable in a kernel is thread ID independent if and only if execution of the instruction or the value of the variable is not affected by the value of thread ID, regardless of it being direct or indirect. Therefore, execution traces of thread ID independent instructions coincide in all threads. If all barrier instructions are thread ID independent, we can ensure freedom from barrier divergence. The definition of thread ID independence is sound, but is not complete [3]. Thus, there exists a program that never causes barrier divergence, yet is not thread ID independent. Such a program cannot be verified by GPUCSL.

We formalize thread ID independence as a type system (Figs. 9, 10, 11). This type system is based on noninterference [8]. In thread ID independence, variables, arithmetic expressions, Boolean expressions, and commands are typed as either type Hi or type Lo. Type Lo means thread ID independence. $\Gamma : \text{Var} \to \{\text{Hi}, \text{Lo}\}$ is a typing environment.

$$\frac{}{\Gamma \vdash \mathtt{tid} : \mathsf{Hi}} \qquad \frac{x \neq \mathtt{tid}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathsf{Lo}} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 + E_2 : \tau_1 \sqcup \tau_2}$$

Fig. 9 Typing rules for expressions

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 = E_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \leq E_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash B_1 : \tau_1 \quad \Gamma \vdash B_2 : \tau_2}{\Gamma \vdash B_1 \&\& B_2 : \tau_1 \sqcup \tau_2} \qquad \frac{\Gamma \vdash B : \tau}{\Gamma \vdash !B : \tau}$$

Fig. 10 Typing rules for Boolean expressions

$$\text{Ty-skip}\frac{}{\Gamma \vdash \mathbf{skip} : \tau} \qquad \text{Ty-Read}\frac{\Gamma \vdash E : \tau \quad \tau \sqcup \tau' \sqsubseteq \Gamma(x)}{\Gamma \vdash x := [E] : \tau'}$$

$$\text{Ty-Write}\frac{}{\Gamma \vdash [E_1] := E_2 : \tau} \qquad \text{Ty-Assign}\frac{\Gamma \vdash E : \tau \quad \tau \sqcup \tau' \sqsubseteq \Gamma(x)}{\Gamma \vdash x := E : \tau'}$$

$$\text{Ty-Seq}\frac{\Gamma \vdash C_1 : \tau \quad \Gamma \vdash C_2 : \tau}{\Gamma \vdash C_1; C_2 : \tau} \qquad \text{Ty-If}\frac{\Gamma \vdash B : \tau' \quad \Gamma \vdash C_1 : \tau \sqcup \tau' \quad \Gamma \vdash C_2 : \tau \sqcup \tau'}{\Gamma \vdash \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 : \tau}$$

$$\text{Ty-While}\frac{\Gamma \vdash B : \tau' \quad \Gamma \vdash C : \tau \sqcup \tau'}{\Gamma \vdash \mathbf{while}\ B\ \mathbf{do}\ C : \tau} \qquad \text{Ty-Barrier}\frac{}{\Gamma \vdash \mathbf{barrier}_b : \mathsf{Lo}}$$

Fig. 11 Typing rules for commands

Fig. 9 shows the typing rules for arithmetic expressions. An expression has type Hi if an expression contains a variable with type Hi; otherwise, it has type Lo. $\tau_1 \sqcup \tau_2$ is defined as follows.

$$\tau_1 \sqcup \tau_2 = \begin{cases} \mathsf{Lo} & (\tau_1 = \mathsf{Lo} \wedge \tau_2 = \mathsf{Lo}) \\ \mathsf{Hi} & (\text{otherwise}) \end{cases}$$

The typing rules for Boolean expressions are defined similarly (Fig. 10). Fig. 11 shows the typing rules for commands. If a command C has type Lo, C is ensured to be executed in a context that is independent from thread ID. $\tau_1 \sqsubseteq \tau_2$ is defined as $\tau_1 = \mathsf{Lo} \vee \tau_2 = \mathsf{Hi}$. The rules Ty-Read and Ty-Assign prevent the values of variables with type Lo from depending on the values of variables with type Hi. The rules Ty-While and Ty-If prevent commands with type Lo from appearing in the bodies of conditional statements whose conditional expression has type Hi. By forcing barrier instructions to have type Lo, we can ensure that barrier synchronization always succeeds (Ty-Barrier).

3.3 Soundness

The definition of soundness of GPUCSL is based on Vafeiadis [13]. The main differences from the Vafeiadis' soundness are in the conditions on barrier synchronization. To define the soundness of GPUCSL, we define a predicate Gsafe_n. Gsafe_n means "the execution is safe for at least n steps". Here, safety means that (i) if the execution terminates, its memory state satisfies the postcondition, (ii) the execution does not abort, and (iii) barrier divergence does not occur in the execution.

Definition 1 (The Gsafe predicate)**.** $\mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$ is defined as follows:

$\mathsf{Gsafe}_0(\overline{C}, \overline{s}, h, Q, \Gamma)$ always holds.

$\mathsf{Gsafe}_{n+1}(\overline{C}, \overline{s}, h, Q, \Gamma)$ holds if and only if

(1) $\overline{C} = \overline{\mathbf{skip}} \Rightarrow \overline{s}, h \models Q$

(2) For all pheaps h_F, if $h \perp h_F$, then $(\overline{C}, \overline{s}, h \uplus h_F) \not\rightarrow_G \mathbf{abort}$

(3) If $\forall i \in Tid, \mathsf{wait}(C_i) = (b_i, C_i')$, then $\forall i, j \in Tid$, $\Gamma \models s_i =_L s_j$ and $b_i = b_j$

(4) For all pheaps h_F, if $h \perp h_F$ and $(\overline{C}, \overline{s}, h \uplus h_F) \rightarrow_g (\overline{C'}, \overline{s'}, h')$ holds, then there exists a pheap h'' such that $h' = h'' \uplus h_F$ and $\mathsf{Gsafe}_n(\overline{C'}, \overline{s'}, h'', Q, \Gamma)$

Here, $\Gamma \models s =_L s'$ is defined as $\forall x, \Gamma(x) = \mathsf{Lo} \Rightarrow s(x) = s'(x)$. $\overline{s}, h \models Q$ is also defined as $\exists s', (\forall i \in Tid, \Gamma \models s_i =_L s') \wedge s', h \models Q$

We define the semantics of $\{P\}\ C\ \{Q\}$ by using the Gsafe predicate as follows.

Definition 2. $\Gamma \models_{par} \{P\}\ C\ \{Q\}$ is defined as follows. For all natural numbers n, stack sequences $\overline{s}$, and pheaps h, if $\overline{s}, h \models P$ and $\forall i, s_i(\mathtt{tid}) = i$, then $\mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$. Here, $\overline{C}$ is a sequence of T Cs.

To prove the soundness of the parallel rule, we define the soundness of rules for sequential execution. We first define the Tsafe predicate like the Gsafe predicate; then we define the semantics of the $\{P\}\ C\ \{Q\}$ for sequential execution in the same way as above.

Definition 3 (Tsafe predicate)**.** $\mathsf{Tsafe}_{i,n}(C, s, h, Q, BS)$ is defined as follows.

$\mathsf{Tsafe}_{i,0}(C, s, h, Q, BS)$ always holds.

$\mathsf{Tsafe}_{i,n+1}(C, s, h, Q, BS)$ holds if and only if

(1) $C = \mathbf{skip} \Rightarrow s, h \models Q$

(2) For all pheaps h_F, if $h \perp h_F$ and $\mathsf{hdef}(h \uplus h_F)$ hold, then $(C, s, h \uplus h_F) \not\rightarrow_T \mathbf{abort}$

(3) $\mathsf{writes}(C, s) \neq \perp \Rightarrow \exists v, h(\mathsf{writes}(C, s)) = (1, v)$

(4) For all pheaps h_F which satisfies $h \perp h_F$ and $\mathsf{hdef}(h \uplus h_F)$, if $(C, s, h \uplus h_F) \rightarrow_t (C', s', h')$ hold, then there exists a pheap h'' such that $h' = h'' \uplus h_F$ and $\mathsf{Tsafe}_{i,n}(C', s', h'', Q, BS)$

(5) If $\mathsf{wait}(C) = (b, C')$, there exist pheaps h_P and h_F such that

- $h_P \perp h_F$, $h = h_P \uplus h_F$ and $s, h_P \models BS(i, b)_{pre}$ hold, and
- for all pheaps h_Q which satisfy $h_Q \perp h_F$ and $s, h_Q \models BS(i, b)_{post}$, $\mathsf{Tsafe}_{i,n}(C', s, h_Q \uplus h_F, Q, BS)$

The predicate hdef in (2) and (4) means that the pheap has read and write permission for all addresses in its domain. If a pheap h satisfies $\mathsf{hdef}(h)$, h can be used as a heap. The function writes used in (3) is an auxiliary function which returns the address written to in the next step of the execution, if the command executes a write instruction in the next step.

$$\mathsf{writes}([E_1] := E, s) = s(E_1)$$
$$\mathsf{writes}(C_1; C_2, s) = \mathsf{writes}(C_1, s)$$
$$\mathsf{writes}(C, s) = \bot \text{ (otherwise)}$$

Definition (5) is for when a barrier instruction is executed in the next step. This means that when a thread reaches a barrier, each thread returns proper resources according to the barrier specification and is safe with the distributed resources.

We define the semantics of $\{P\}\ C\ \{Q\}$ for sequential execution by using the Tsafe predicate as follows.

Definition 4. $BS, i \models_{seq} \{P\}\ C\ \{Q\}$ is defined as follows. For all natural numbers n, stacks s and pheaps h, if $s, h \models P$, then $\mathsf{Tsafe}_{i,n}(C, s, h, Q, BS)$.

4. Proof of soundness

We proved the soundness of GPUCSL by using Coq. This proof is based on Vafeiadis' proof with Coq [12], and it is about 4000 LOC. Because the Vafeiadis' CSL has many differences compared with GPUCSL, we formalized GPUCSL without reusing Vafeiadis' Coq proof. Our proof is available at `http://prg.is.titech.ac.jp/ja/projects/gpucsl/`.

Because Vafeiadis' system uses atomic instructions as a synchronization model, we changed the definition of soundness to fit barrier synchronization. We applied Vafeiadis' proof to the soundness of the inference rules for sequential execution. We proved the soundness of thread ID independence and showed the soundness of the rules for parallel execution by applying the Vafeiadis' proof to it, but our proof is different from Vafeiadis' in the point that it uses the soundness of thread ID independence.

In the following, we outline the soundness proof. The following theorem is the soundness of GPUCSL.

Theorem 5 (Soundness of GPUCSL)**.** $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\} \Rightarrow \Gamma \models_{par} \{P\}\ C\ \{Q\}$

To prove Theorem 5, we first prove the soundness of the inference rules for sequential execution. Next, we show the soundness of thread ID independence (barrier divergence does not occur, and values in all thread ID independent variables coincide in all threads when reaching a barrier instruction or terminating). Finally, we show the soundness of the inference rule for parallel execution by using these two lemmas.

In this section, we assume the barrier specification BS is precise and satisfies $\forall b,\ \bigstar_{i \in Tid} BS(i,b)_{pre} \Rightarrow \bigstar_{i \in Tid} BS(i,b)_{post}$

4.1 Soundness of inference rules for sequential execution

The following lemma is soundness of the inference rules for sequential execution.

Lemma 6 (Soundness for sequential execution)**.** $BS, i \vdash_{seq} \{P\}\ C\ \{Q\} \Rightarrow BS, i \models_{seq} \{P\}\ C\ \{Q\}$

The proof is done by induction on derivations of $BS, i \vdash_{seq} \{P\}\ C\ \{Q\}$, and in each case of its derivation, we unfold the definition of $\models_{seq}$ and prove the case by induction on the index n of the Tsafe predicate. The cases other than the barrier rule can be proved as Vafeiadis did. So, in the following, we prove the case of the barrier rule. First, we prove the following lemma.

Lemma 7. For all n, BS, i, s, h, if $s, h \models P$, then $\mathsf{Tsafe}_{i,n}(\mathbf{skip}, s, h, P, BS)$

Proof. If $n = 0$, this is trivial. So consider when $n + 1$. We prove each condition of Tsafe. (1) can be proved by $s, h, \models P$. The other cases can also be easily proved. □

Lemma 8 (Soundness of barrier rule)**.** For all BS, i, b, s, h, n, if $s, h \models BS(i,b)_{pre}$, then $\mathsf{Tsafe}_{i,n}(\mathbf{barrier}_b, s, h, BS(i,b)_{post}, BS)$

Proof. By induction on n. The case of $n = 0$ is trivial. Consider the case of $n = k + 1$. We prove each condition of Tsafe. (1), (2), (3), and (4) are trivial. Next, we prove (5). $\mathsf{wait}(\mathbf{barrier}_b) = (b, \mathbf{skip})$ holds. We choose h as h_p, and a pheap that satisfies $\forall l, h_F(l) = \bot$ as h_F. $h_p \bot h_F$, $h = h_p \uplus h_F$ and $s, h_p \models BS(i,b)_{pre}$ trivially hold. Assume that h_Q is an arbitrary pheap which satisfies $h_Q \bot h_F$ and $s, h_Q \models BS(i,b)_{post}$. From $h_Q \uplus h_F = h_Q$, we only need to show $\mathsf{Tsafe}_{i,k}(\mathbf{skip}, s, h_Q, BS(i,b)_{post}, BS)$, and this follows from $s, h_Q \models BS(i,b)_{post}$ and Lemma 7 □

4.2 Soundness of thread ID independence

We define a predicate $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ that represents the initial state of a kernel.

Definition 9. $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ holds if and only if

- $\exists C_{init}\ \tau, (\forall i \in Tid, C_i = C_{init}) \wedge \Gamma \vdash C_{init} : \tau$
- $\forall i, j \in Tid, \Gamma \models s_i =_L s_j$
- $\biguplus_{i \in Tid} h_i$ is defined
- $\forall i \in Tid, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$

The following lemma is the soundness of thread ID independence.

Lemma 10. If $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ and $(\overline{C}, \overline{s}, \biguplus_{i \in Tid} h_i) \rightarrow_g^* (\overline{C'}, \overline{s'}, h')$, then the following conditions hold.

(1) If $\forall i \in Tid, C'_i = \mathbf{skip}$, then $\forall i, j \in Tid, \Gamma \models s'_i =_L s'_j$

(2) If $\forall i \in Tid, \mathsf{wait}(C'_i) = (b_i, C''_i)$, then $\forall i, j \in Tid, b_i = b_j \wedge s'_i =_L s'_j$

(1) means that when all threads terminate, all values of variables with type Lo coincide in all threads, and (2) means that when all threads reach barriers, all values of variables with type Lo and all indices of the barriers (b_i) coincide in all threads.

To prove Lemma 10, we first show the following lemma.

Lemma 11 (Non-interference)**.** If $\Gamma \vdash C : \tau$, $s_1 =_L s_2$, $h_1 \bot h_2$, $(C, s_1, h_1) \rightarrow_p^* (C_1, s'_1, h'_1)$ and $(C, s_2, h_2) \rightarrow_p^* (C_2, s'_2, h'_2)$, then the following conditions hold.

(1) $\forall i \in \{1,2\}, \mathsf{wait}(C_i) = \mathbf{skip} \Rightarrow \forall i, j \in \{1,2\}, s'_i =_L s'_j$

(2) $\forall i \in \{1,2\}, \mathsf{wait}(C_i) = (b_i, C'_i) \Rightarrow \forall i, j \in \{1,2\}, b_i = b_j \wedge s'_i =_L s'_j$

Here, $\rightarrow_p$ are extended semantics of $\rightarrow_t$ in order to use pheap instead of heap, and they are defined as follows.

Definition 12. $(C, s, h) \rightarrow_p (C', s', h')$ holds if and only if

the following hold.

(1) $\mathsf{writes}(C,s) \neq \bot \Rightarrow \exists v, h(\mathsf{writes}(C,s)) = (1,v)$

(2) $\mathsf{reads}(C,s) \neq \bot \Rightarrow \exists v,p, h(\mathsf{reads}(C,s)) = (p,v)$

(3) There exists a pheap h_F such that $h \bot h_F$, $\mathsf{hdef}(h \uplus h_F)$, and $(C,s,h \uplus h_F) \rightarrow_t (C',s',h' \uplus h_F)$

reads is an auxiliary function that returns the address which will be read from in the next step if the command executes a read instruction in the next step.

$$\begin{aligned}\mathsf{reads}(x := [E], s) &= s(E)\\ \mathsf{reads}(C_1; C_2, s) &= \mathsf{reads}(C_1, s)\\ \mathsf{reads}(C, s) &= \bot \text{ (otherwise)}\end{aligned}$$

We omit the proof of Lemma 11.

To apply Lemma 11 to the whole kernel semantics, we prove the following lemma.

Lemma 13. If $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ and $(\overline{C}, \overline{s}, \biguplus_{i \in Tid} h_i) \rightarrow_g^* (\overline{C'}, \overline{s'}, h')$, then there exist $\overline{C''}$, $\overline{s''}$, and $\overline{h''}$ such that $\mathsf{initConf}(\overline{C''}, \overline{s''}, \overline{h''}, \Gamma, BS)$ and $\forall i \in Tid, (C_i'', s_i'', h_i'') \rightarrow_p^* (C_i', s_i', h_i')$

Proof. By induction on $\rightarrow_g^*$ □

By using Lemma 11 and Lemma 13, we can prove Lemma 10 as follows.

Proof. By applying Lemma 13 to the hypotheses, there exist $\overline{C''}, \overline{s''}$ and $\overline{h''}$ such that $\mathsf{initConf}(\overline{C''}, \overline{s''}, \overline{h''}, \Gamma, BS)$ and $\forall i \in Tid, (C_i'', s_i'', h_i'') \rightarrow_p^* (C_i', s_i', h_i')$. Here, we choose a command that satisfies $\forall i \in Tid, C_i'' = C_{init}$ as C_{init}. For all i and j, we apply Lemma 11 to $(C_{init}, s_i'', h_i'') \rightarrow_p^* (C_i', s_i', h_i)$ and $(C_{init}, s_j'', h_j'') \rightarrow_p^* (C_j', s_j', h_j')$. This proves the lemma. □

4.3 Soundness of CSL on parallel execution

We prove Theorem 5 by using Lemma 6 and Lemma 10. First, we prove a lemma.

Lemma 14. Assume the following conditions hold.

(a) $h = \biguplus_{i \in Tid} h_i$

(b) $\forall i \in Tid, \mathsf{Tsafe}_{i,n}(C, s_i, h_i, Q_i, BS)$

(c) $\mathsf{initConf}(\overline{C_{init}}, \overline{s_{init}}, \overline{h_{init}}, \Gamma, BS)$

(d) $(\overline{C_{init}}, \overline{s_{init}}, \overline{h_{init}}) \rightarrow_g^* (\overline{C}, \overline{s}, h)$

(e) $\bigstar_{i \in Tid} Q_i \Rightarrow Q$

Then $\mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$.

Proof. By induction on n. The case of $n = 0$ is trivial. Consider the case of $n = n' + 1$. We prove each condition of Gsafe_k.

(1) Assume $\forall i \in Tid$ and $C_i = \mathbf{skip}$. By (b), $\forall i \in Tid$, $s_i, h_i \models Q_i$. By applying Lemma 10 to (c) and (d), we get $\forall i,j \in Tid, s_i =_L s_j$. Therefore, $\overline{s}, h \models \bigstar_{i \in Tid} Q_i$. From (e), $\overline{s}, h \models Q$.

(2) For all i', we define $h' = (\biguplus_{i \in Tid \wedge i \neq i'} h_i) \uplus h_F$. From case (2) of (b) with $i = i'$ and $h_F = h'$, $(C_i, s_i, h \uplus h_F) \nrightarrow_t \mathbf{abort}$, since $h_i \uplus h' = h \uplus h_F$. Because i' is an arbitrary thread ID, $(\overline{C}, \overline{s}, h \uplus h_F) \nrightarrow_g \mathbf{abort}$.

(3) Similarly to (1), this can be proved by applying Lemma 10 to the hypotheses.

(4) Assume $(\overline{C}, \overline{s}, h \uplus h_F) \rightarrow_g (\overline{C'}, \overline{s'}, h')$, and let us use case analysis on the derivation. If the derivation is (G-Step), let k be the thread ID of the executed thread, and $(C_k, s_k, h \uplus h_F) \rightarrow_t (C_k', s_k', h')$. Here, $C_i' = C_i$ and $s_i' = s_i$ hold for i other than k. We choose $h_F' = (\biguplus_{i \in Tid, i \neq k} h_i) \uplus h_F$, and $h \uplus h_F = h_k \uplus h_F'$ holds. From (4) of (b) with $i := k$, there exists h_k'' such that $h' = h_k'' \uplus h_F'$ and $\mathsf{Tsafe}_{k,n'}(C_k', s_k', h_k'', Q_k)$. For all i other than k, we choose h_i as h_i'', and from (b), $\mathsf{Tsafe}_{i,n'}(C_i', s_i', h_i'', Q_i)$ holds. Therefore, $\forall i \in Tid, \mathsf{Tsafe}_{i,n'}(C_i', s_i', h_i'', Q_i)$. Let h'' be $\biguplus_{i \in Tid} h_i''$; then $h' = h'' \uplus h_F$. By applying the induction hypothesis, we get $\mathsf{Gsafe}_{n'}(\overline{C'}, \overline{s'}, h'', Q, \Gamma)$. The G-Barrier case can be proved similarly.

□

We prove Theorem 5 by using Lemma 14.

Proof. From $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}$, the following conditions hold.

(1) $P \Rightarrow \bigstar_{i \in Tid} P_i$

(2) $\bigstar_i Q_i \Rightarrow Q$

(3) $\forall i \in tid, BS, i \models \{P_i\}\ C_i\ \{Q_i\}$

(4) $\Gamma \vdash C : \tau$

(5) $\forall b,\ \bigstar_{i \in Tid} BS(i,b)_{pre} \Rightarrow \bigstar_{i \in Tid} BS(i,b)_{post}$

Here, we prove that for all $\overline{s}$ and h, if $\forall i \in Tid$, $s_i(\mathtt{tid}) = i$, $\forall i,j \in Tid, s_i =_L s_j$ and $\overline{s}, h \models P$, then $\forall n, \mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$. By (1), $\exists \overline{h}, h = \uplus_i h_i$ and $\forall i, s_i, h_i \models P_i$. From (3), $\forall i, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$. Trivially, $(\overline{C}, \overline{s}, h) \rightarrow_g^* (\overline{C}, \overline{s}, h)$. Applying Lemma 14 to these conditions concludes the proof. □

5. Differences from Blom's CSL

In this section, we compare Blom's research and ours in terms of their inference rules and soundness proofs.

5.1 Differences between inference rules

As differences between Blom's CSL and GPUCSL, we can point out (i) the different forms of the assertions and (ii) the absence of the frame rule.

(i) Blom's inference rules are not suitable for proving its soundness in Coq. An assertion in Blom's CSL consists of an assertion on resources and an assertion on functions. A specification of Blom's CSL takes the form $\{R_{pre}, P_{pre}\}\ C\ \{R_{post}, P_{post}\}$. R_{pre} and R_{post} are assertions on resources, and P_{pre}, and P_{post} are assertions on functions.

Fig. 12 is the write rule of Blom's CSL. The predicate $\mathsf{LPerm}(e, \mathsf{rw})$ in assertions on resources is equivalent to $\exists e', e \mapsto^1 e'$ of GPUCSL. The expression $L[e_1]$ in assertions on functions means the value that the address e_1 points to. All inference rules of Blom's CSL require that all addresses appearing in assertions on functions are referred to by assertions on resources as a premise. This condition is relatively complex, and hence, it is considered to complicate the soundness proof.

$$\text{Write}\ \frac{}{\{R \star \mathsf{LPerm}(e_1, \mathsf{rw}), P[L[e_1] := e_2]\}\mathsf{wrloc}(e_1, e_2)\{R \star \mathsf{LPerm}(e_1, \mathsf{rw}), P\}}$$

Fig. 12 Write rule of Blom's CSL

(ii) While Blom's CSL does not have a frame rule, GPUCSL has one since it is designed to be similar to standard CSL. The frame rule is needed to describe specifications of program functions in a modular manner, and as we describe later, we cannot simply apply Blom's proof to CSLs which have the frame rule.

5.2 Problems with Blom's soundness proof

We show that (i) it is difficult to add the frame rule to Blom's proof and (ii) Blom's proof lacks an appropriate premise for thread ID independence.

(i) Blom's soundness proof depends on the following Lemma 15 [*1].

Lemma 15. If $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}$, $\overline{s}, h \models P$, $(\overline{C}, \overline{s}, h) \rightarrow_g^* (\overline{C'}, \overline{s'}, h')$ and $\forall i \in Tid, \mathsf{wait}(C_i') = (b, C_i'')$, then $\overline{s'}, h' \models \bigstar_{i \in Tid} BS(i, b)_{pre}$

This lemma means that when a kernel C satisfying $\{P\}\ C\ \{Q\}$ reaches a barrier instruction, the memory state when reaching the barrier satisfies the precondition of the barrier specification. However, this lemma does not hold on CSLs with the frame rule. Consider the following kernel C.

```
1 [a+tid] = tid;
2 barrier0;
3 [a+tid] = tid;
```

By using GPUCSL, we can derive $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}$ with the following assertions.

- $P := \bigstar_{i \in Tid}(\exists v, \mathtt{a} + i \mapsto v)$
- $Q := \bigstar_{i \in Tid}(\mathtt{a} + i \mapsto i)$
- $P_i := \exists v, \mathtt{a} + i \mapsto v$
- $Q_i := \mathtt{a} + i \mapsto i$
- $\forall i, BS(i, 0) := \mathbf{emp}$

We can prove $\forall i, BS, i \vdash_{seq} \{P_i \wedge \mathtt{tid} = i\}\ C\ \{Q_i\}$ as follows.

```
1  {∃v, a + i ↦ v ∧ tid = i} ⇒
2  {a + tid ↦ v ∧ tid = i}
3    [a+tid] = tid;
4  {a + tid ↦ tid ∧ tid = i} ⇒
5  {emp ⋆ (a + tid ↦ tid ∧ tid = i)}
6    barrier0;
7  {emp ⋆ (a + tid ↦ tid ∧ tid = i)} ⇒
8  {a + tid ↦ tid ∧ tid = i}
9    [a+tid] = tid;
10 {a + tid ↦ tid ∧ tid = i} ⇒
11 {a + i ↦ i}
```

Note that we use the frame rule on line 6. Here, the global heap h satisfies $\forall i \in Tid, h(i) = i$ when reaching the barrier instruction, so apparently it does not satisfy $\overline{s}, h \models \bigstar_{i \in Tid}\mathbf{emp} = \mathbf{emp}$. Accordingly, Blom's soundness proof is not suitable for GPUCSL

(ii) We prove thread ID independent kernels do not suffer from barrier divergence in Lemma 10. We assume a condition that is not mentioned by Blom et al.: $\forall i \in Tid, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$. This condition means the kernel is free of data races, and this condition cannot be omitted. This is because there is a counterexample kernel that is thread ID independent, but for which data races and barrier divergence occur:

```
1 x := [a];
2 [a] := tid;
3 if (x == 0) {
4   barrier0;
5 }
```

We can derive $\Gamma \vdash C : \mathsf{Lo}$ under a type environment Γ that satisfies $\Gamma(\mathtt{a}) = \mathsf{Lo} \wedge \Gamma(\mathtt{x}) = \mathsf{Lo}$. However, because the result of evaluating the conditional expression on line 3 depends on scheduling, barrier divergence can occur with this kernel.

6. Application

As a merit of defining GPUCSL by using Coq, we can verify kernels on Coq. Here, we have verified the kernel shown in Fig. 2 (rotate) on Coq. The proof is about 1000 LOC and is available at `http://prg.is.titech.ac.jp/ja/projects/gpucsl/`. Here, we show a sketch of the proof done by Coq. rotate moves the i-th element of array a of length T to the $((i+1) \bmod T)$-th location. We specify rotate as follows.

$$\{\mathsf{is\_array}(\mathtt{a}, T, f)\}$$
$$\mathsf{rotate}$$
$$\{\mathsf{is\_array}(\mathtt{a}, T, \lambda i.f((i-1) \bmod T))\}$$

The predicate $\mathsf{is\_array}(arr, n, f)$ means that arr is an array of length n, and the i-th element is initialized by $f(i)$. Here, arr is an expression, n is a natural number, and f is a function from a natural number to an expression. is_array is defined as follows.

$$\begin{aligned}\mathsf{is\_array}(arr, 0, f) &= \mathbf{emp}\\ \mathsf{is\_array}(arr, n+1, f) &= ((arr + n) \mapsto f(n)) \star \\ &\quad \mathsf{is\_array}(arr, n, f)\end{aligned}$$

Now let us prove each premise of the parallel rule. First, we prove $\Gamma \vdash \mathsf{rotate} : \mathsf{Lo}$. This is easily done by taking Γ that satisfies $\Gamma(x) = \mathsf{Hi}$ for all x other than a. Next, we choose each assertion as follows.

- $P_i = (\mathtt{a} + i \mapsto f(i))$
- $Q_i = (\mathtt{a} + (i+1) \bmod T) \mapsto f(i)$
- $BS(i, 0)_{pre} = (\mathtt{a} + i \mapsto f(i))$
- $BS(i, 0)_{post} = (\mathtt{a} + (i+1) \bmod T \mapsto f((i+1) \bmod T))$

i and T appearing in each predicate are constants, so from $\Gamma(\mathtt{a}) = \mathsf{Lo}$, these predicates are thread ID independent. The conditions on assertions ($P \Rightarrow \bigstar_{i \in Tid} P_i$, $\bigstar_{i \in Tid} Q_i \Rightarrow Q$ and $\bigstar_{i \in Tid} BS(i, 0)_{pre} \Rightarrow \bigstar_{i \in Tid} BS(i, 0)_{post}$) can be easily proved. So, we will prove that for all i, $BS, i \vdash_{seq} \{P_i \wedge \mathtt{tid} = i\}\ \mathsf{rotate}\ \{Q_i\}$ by case analysis on $i < T - 1$

[*1] Lemma 15 corresponds to the following statement in [3]. "Since the barrier resources properly divide the group resources, the resources required by the second part of the trace are available."

1 $\{\mathtt{a} + i \mapsto f(i) \land \mathtt{tid} = i\} \Rightarrow$
2 $\{\mathtt{a} + \mathtt{tid} \mapsto f(i) \land \mathtt{tid} = i\}$
3 `x := [a + tid];`
4 $\{\mathtt{a} + \mathtt{tid} \mapsto f(i) \land \mathtt{tid} = i \land \mathtt{x} = f(i)\} \Rightarrow$
5 $\{\mathtt{a} + i \mapsto f(i) \land \mathtt{tid} = i \land x = f(i)\}$
6 $\mathtt{barrier}_0$`;`
7 $\{\mathtt{a} + (i+1) \bmod T \mapsto f((i+1) \bmod T) \land \mathtt{tid} = i \land \mathtt{x} = f(i)\} \Rightarrow$
8 $\{\mathtt{a} + (i+1) \mapsto f(i+1) \land \mathtt{tid} = i \land \mathtt{x} = f(i)\}$
(By $i < T - 1 \Rightarrow (i+1) \bmod T = i + 1$)
9 `t = tid + 1;`
10 `if (tid == T - 1) {`
11 `t = 0`
12 `}`
13 $\{\mathtt{a} + (i+1) \mapsto f(i+1) \land \mathtt{tid} = i \land \mathtt{x} = f(i) \land \mathtt{t} = \mathtt{tid} + 1\} \Rightarrow$
14 $\{\mathtt{a} + \mathtt{t} \mapsto f(i+1) \land \mathtt{tid} = i \land \mathtt{x} = f(i) \land \mathtt{t} = \mathtt{tid} + 1\}$
15 `[a + t] := x;`
16 $\{\mathtt{a} + \mathtt{t} \mapsto f(i) \land \mathtt{tid} = i \land \mathtt{x} = f(i) \land \mathtt{t} = \mathtt{tid} + 1\} \Rightarrow$
17 $\{(\mathtt{a} + (i+1)) \bmod T \mapsto f(i)\}$

Fig. 13 Proof of $BS, i \vdash_{seq} \{P_i \land \mathtt{tid} = i\}$ rotate $\{Q_i\}$

and $i = T - 1$. We can prove this when $i < T - 1$ in the same way as in Fig. 13 (we can prove the case of $i = T - 1$ in a similar way).

7. Related work

GPUVerify [2] is a verifier for kernels which detects data races and barrier divergence. Betts et al. designed SDV semantics which can describe these conditions, and GPUVerify verifies kernels under the SDV semantics. By using `assert` and `assume` statements, GPUVerify can also verify specifications of kernels as well as GPUCSL. However, the specifications which can be proved in GPUVerify are restricted to those which can be solved by the SMT solvers used by GPUVerify.

Kojima et al. proposed a Hoare logic for single instruction multiple threads (SIMT) programs and proved the soundness and relative completeness of the logic [7]. SIMT semantics force all threads to execute a every step of a program simultaneously. GPUCSL assumes all threads execute a program in arbitrary order.

Vafeiadis devised a concise proof of the soundness of CSL and used it to prove the soundness of FPCSL [13]. He also proved soundness by using Coq and Isabelle/HOL. We applied Vafeiadis' proof to the soundness of GPUCSL.

Affeldt et al. formalized the separation logic for verifying TSL packet processing programs written in the C language by Coq/SSReflect [1]. Their separation logic can represent detailed specifications such as the alignment of data structures and the behavior of the `sizeof` operator. Our language omits composite data types and only has the integer data type.

Hobor et al. proposed a CSL for languages which have Pthread-like barrier synchronization [6]. He chose Cminor, which is an intermediate representation of the CompCert compiler, as the object language, and proved soundness of their CSL by using Coq. As differences from GPUCSL, we point out that dynamic thread creation is allowed in their CSL and they do not verify freedom from barrier divergences.

8. Conclusion and future work

We designed a CSL for verifying GPGPU kernels and proved its soundness by using Coq. The inference rules are designed based on Blom's CSL and Vafeiadis' CSL, and we proved its soundness by applying Vafeiadis' soundness proof. We also proved the soundness of thread ID independence, which was not given by Blom et al. Through the design and the proof, we showed that (i) Blom's proof cannot be straightforwardly applied to CSLs with the frame rule, and (ii) the soundness of thread ID independence depends on the kernels being free of data races.

8.1 Towards more precise GPGPU semantics

There are important CUDA features that are not considered in our CUDA subset: thread blocks and warps. In CUDA, the set of all threads are divided into units of thread blocks. Moreover, each thread block is divided into units of warps. A thread block is a set of warps, and a warp is a set of threads. Each thread block has its own memory that runs fast (shared memory). Barrier synchronization is done in units of thread blocks. Threads which belong to the same warp are executed in an SIMT manner.

The semantics of the WhileB language are such that all threads belong to the same thread block. However, we can easily add a thread block feature to GPUCSL. First, we would add an inference rule to GPUCSL which distributes the kernel precondition to each thread block, verify each thread block by using the parallel rule, and aggregate the postconditions of each thread block into the kernel postcondition. Then, we would give type Lo to the variable `bid`, which means the number of the threads in each thread block. Finally, we would introduce $\mapsto$ operator between shared memories.

In contrast, because GPUCSL was proved on arbitrary scheduling, it is also sound under semantics having the warp feature. However, kernels that omit barrier synchronization and assume SIMT execution are considered racy, so we cannot use GPUCSL to verify these kernels. These omissions of barrier synchronization are optimization techniques for GPGPU [5], so we should extend CSL to enable these kernels to be verified.

8.2 Design of GPGPU kernel verification library in Coq

We proved the soundness of GPUCSL by using Coq. We can use this proof to verify GPGPU kernels in Coq. We verified the rotate kernel in Coq and it required many lemmas. A future task would be to design lemma libraries and tactics that would make it easier to verify kernels.

Acknowledgments We wish to thank Atsushi Igarashi and Kensuke Kojima of Kyoto University for helpful advice on this research, the reviewer for many important comments on this paper, and our laboratory members for their discussions on this research.

References

[1] Affeldt, R. and Sakaguchi, K.: An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing, *Journal of Formalized Reasoning*, Vol. 7, No. 1 (2014).

[2] Betts, A., Chong, N., Donaldson, A., Qadeer, S. and Thomson, P.: GPUVerify: A Verifier for GPU Kernels, *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 113–132 (2012).

[3] Blom, S., Huisman, M. and Mihelčić, M.: Specification and Verification of GPGPU programs, *Science of Computer Programming*, Vol. 95, Part 3, pp. 376 – 388 (2014).

[4] Bornat, R., Calcagno, C., O'Hearn, P. and Parkinson, M.: Permission Accounting in Separation Logic, *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 259–270 (2005).

[5] Harris, M.: Optimizing parallel reduction in CUDA (2007).

[6] Hobor, A. and Gherghina, C.: Barriers in Concurrent Separation Logic, *Proceedings of 20th European Symposium on Programming*, Lecture Notes in Computer Science, Vol. 6602, pp. 276–296 (2011).

[7] Kojima, K. and Igarashi, A.: A Hoare Logic for SIMT Programs, *Proceedings of 11th Asian Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, Vol. 8301, pp. 58–73 (2013).

[8] Myers, A.: Proving Noninterference for a While-Language Using Small-Step Operational Semantics (2011). Tutorial Note for the Marktoberdorf Summer School on Logics and Languages for Reliability and Security.

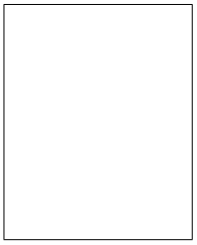
[9] NVIDIA: CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` (2015).

[10] O'Hearn, P. W.: Resources, Concurrency and Local Reasoning, *CONCUR 2004 - Concurrency Theory*, Lecture Notes in Computer Science, Vol. 3170, pp. 49–67 (2004).

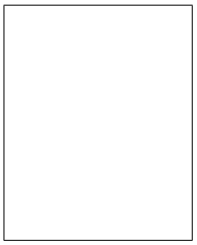
[11] The Coq Development Team: The Coq Proof Assistant Reference Manual (2014).

[12] Vafeiadis, V.: Concurrent Separation Logic Soundness, `https://www.mpi-sws.org/~viktor/cslsound/`.

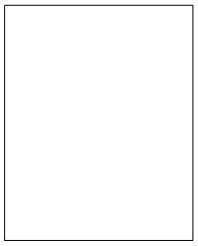
[13] Vafeiadis, V.: Concurrent Separation Logic and Operational Semantics, *Electronic Notes in Theoretical Computer Science*, Vol. 276, pp. 335–351 (2011).



Izumi Asakura is a master course student at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.S. degree from Tokyo Institute of Technology in 2014. His research interests include programming languages.



Hidehiko Masuhara is a Professor at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1992, 1994 and 1999 respectively. Before joining Tokyo Institute of Technology, he served as an Assistant Professor, Lecturer, and Associate Professor at Graduate School of Arts and Sciences, the University of Tokyo. His research interests include design and implementation of programming languages and software development environments.



Tomoyuki Aotani is an assistant professor at Department of Mathematical and Computing Sciences, Tokyo Institute of Technology. He received his BSc from Hosei University in 2004 and MA and PhD from the University of Tokyo in 2006 and 2009, respectively. His research interests include design of programming languages, program analysis, verification and optimization.