平成27年度 修士論文

並行分離論理に基づくGPGPU向け データ並列DSLコンパイラの形式検証

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 14M37010

朝倉 泉

指導教員

増原英彦教授
南出靖彦教授

平成28年2月29日

概要

高性能な GPGPU プログラムを，map や reduce といったよく知られた並列スケルトンを用いて容易に記述するための言語として，GPGPU 向けデータ並列 DSL (GPUDSL) が知られている．GPUDSL のコンパイラは，処理する配列の型などでパラメタライズされた GPGPU コード (コードテンプレート) を用いてコード生成を行うが，そのようなコード生成の正しさを保証することは容易ではない．その要因として (i) GPGPU の並列モデルや，(ii) 引数に対して生成コードや仕様が変化する点が挙げられる．本研究ではコードテンプレートを GPGPU コードを生成する Coq の関数としてモデル化し，その正しさを Blom らによって与えられた並行分離論理を用いて Coq 上で検証する．提案手法によって map，zip，reduce といった典型的なスケルトンに対応するコードテンプレートの正しさを検証することができた．

謝辞

本研究を進めるにあたり, 研究の方針や論文の構成法など数々の有用な助言を下さった指導教員の増原 英彦教授，副指導教員の南出 靖彦教授，及び青谷 知幸助教に深く感謝致します．また，本研究に関して多くの有用な助言を下さった京都大学の五十嵐 淳教授，小島 健介氏，奥村 健太郎氏に深く感謝致します．最後に研究や論文に関する有用なコメントをくださった研究室メンバの皆さまに深く感謝いたします.

目 次

第 1 章　はじめに **6**

第 2 章　GPU のための並行分離論理 GPUCSL **8**
2.1 言語 8
2.1.1 構文規則 9
2.1.2 WhileB 言語の操作的意味論 10
2.2 GPUCSL 12
2.2.1 言明の意味論 13
2.2.2 証明規則 14
2.2.3 スレッド ID 独立性 16
2.2.4 健全性 17
2.3 健全性証明 19
2.3.1 逐次実行についての健全性 19
2.3.2 スレッド ID 独立性の健全性 20
2.3.3 並列実行に関する CSL の健全性 21
2.4 Blom の CSL との違い 21
2.4.1 推論規則の違い 21
2.4.2 健全性証明の問題点 22
2.5 応用例 23

第 3 章　GPGPU カーネル検証のためのライブラリ: GPUVeLib **25**
3.1 Coq と Ltac 言語 25
3.2 GPUVeLib 26
3.2.1 自動証明タクティック 26
3.3 手動証明を補助するタクティック 27
3.4 事例研究 28
3.4.1 事例研究 1: map カーネル 28
3.4.2 事例研究 2: reduce カーネルの検証 30
3.4.3 事例研究 3: prescan カーネルの検証 31

第 4 章　メタ言語コードを含むコードテンプレートの検証 **34**
4.1 記法 34
4.2 配列アクセスのためのメタ言語関数 35
4.3 ユーザ関数コード 35
4.4 事例研究 1: map コードテンプレートの検証 36
4.5 事例研究 2: reduce テンプレートの検証 37

第 5 章　関連研究 **41**
5.1 GPGPU のための自動検証器 41
5.2 プログラム論理 41
5.3 GPGPU の意味論 42

5.4 コード生成器の正しさ 42
5.5 安全性と容易性を両立する高性能プログラミング機構 42

第 6 章 結論・今後の課題 **44**
6.1 コンパイラ全体の検証 44
6.2 より高度な CUDA の意味論への対応 45
6.3 コードテンプレートの証明の容易化 45

図目次

2.1 WhileB 言語の構文規則 9
2.2 stencil カーネル 9
2.3 スレッド，スレッドブロック，グリッドの状態 10
2.4 グリッド実行の意味論 10
2.5 ブロック実行の意味論 10
2.6 逐次実行の意味論 11
2.7 Definition of the wait function 11
2.8 式の意味論 11
2.9 並列実行のための証明規則 12
2.10 GPUCSL の論理式の意味論 13
2.11 GPUCSL の証明規則 15
2.12 式に対する型付け規則 15
2.13 LExp に対する型付け規則 16
2.14 ブール式に対する型付け規則 16
2.15 コマンドのための型付け規則 16
2.16 Blom らの CSL の Write 規則 22

3.1 Coq による証明例 26
3.2 hoare_forward の実行例 27
3.3 sep_cancel の実行例 27
3.4 sep_rewrite array_forward の実行例 28

4.1 配列アクセスコードを生成するメタ言語関数の定義 35

第1章　はじめに

本研究は general-purpose computing on GPUs (GPGPU) コードを生成するコードテンプレートに対する検証手法を提案する．

GPGPU とは GPU を用いた高性能計算一般である．GPU が高性能な計算資源を安価に提供するため，多くの目的で利用されている．しかし，高性能な GPGPU プログラムを正しく記述することは容易ではない．GPGPU の代表的な処理系である CUDA C [31] では，データ競合やバリア相違 (barrier divergence) を起こさないようにすることはプログラマの責任である．それに加えて高性能を達成するには，共有メモリなどのメモリ階層を利用し，warp divergence，bank conflict などの並列化阻害要因を避けるために複雑なプログラムを記述する必要がある．

高性能な GPGPU プログラムを簡単に記述するための言語として map や reduce などのよく知られた並列スケルトンを用いたデータ並列 DSL の GPU 向け実現 (GPUDSL) がある．GPUDSL は並列スケルトンを用いて記述されたコードから CUDA C などで記述された GPGPU コードを生成する．

本研究では GPUDSL コンパイラ内におけるコードテンプレートを利用したコード生成に注目する．その理由は，コードテンプレートがユーザにとって宣言的かつ高性能なプログラムを可能にする鍵であるとともに，GPGPU プログラムとして正しさを保証することが容易でない性質を持つためである．コードテンプレートは GPUDSL コンパイラがサポートするスケルトン毎に持つ，その計算を行う GPGPU コードであり，入力や出力の配列の型，スケルトンが受け取る関数 (ユーザ関数) のコンパイル結果 (ユーザ関数コード)，GPU のコア数などでパラメタライズされている．テンプレート中のコードはあらかじめ様々な最適化が施されている．コードテンプレートの正しさ (任意のコードテンプレートが生成する GPGPU コードが正しい計算を行うこと) を保証するための難しさの要因として以下の点が挙げられる．

(1) 生成されるコードやその仕様が引数のユーザ関数コードや配列の型に対して変化するため，通常の GPGPU プログラム向け検証器を用いることが難しい．

(2) テンプレートが生成するコードの計算の正しさをモジュラーに検証するためには，Frame 規則 [34] をもつ健全性が確認された GPGPU 向け並行分離論理 (GPUCSL) が必要になるが，我々が知る限りそのような論理は存在しない．GPGPU 向けプログラム論理として Blom らによって与えら GPUCSL [10] があるが，Blom らの GPUCSL の健全性は非形式的な証明しか与えられておらず，その健全性を確認することが困難である．また Blom らの GPUCSL は Frame 規則を持たず，その拡張可能性，つまり Frame 規則で拡張した GPUCSL が健全であることが明らかでない．

(3) GPGPU のような大量のスレッドが大量のデータを処理するような計算モデルでは，各スレッドの計算パターンが複雑化しやすく，検証に不可欠なループ不変式を与えることが容易ではない．

(1) の解決のために，本研究では定理証明支援器 Coq [35] を用いたコードテンプレートの検証手法を提案する．コードテンプレートを GPGPU コードを生成する Coq の関数として記述し，コードテンプレートの正しさを Coq の定理として証明する．コードテンプレートの正しさとは，生成される任意のコードが正しく振る舞うことであり，本研究ではそれを GPUCSL を用いて検証する．また，Coq 上で GPGPU プログラム検証を補助するための Coq ライブラリ GPUVeLib を作成した．本研究では reduce や prescan といった比較的複雑な同期パターンをもつ GPGPU カーネルの検証を行い GPUVeLib の有用性を確認した．この手法の有用性の確認のために，GPUDSL の 1 つである Accelerate [13] の map コードテンプレート，及び reduce テンプレートの一部の正しさを検証した．

(2) の解決のために，本研究では Blom らの GPUCSL を Frame 規則で拡張した上で Coq で再形式化し，その健全性の証明を新たに与えた．健全性証明の過程で Blom らの証明のスケッチに関して 2 つの欠陥を発見し，それらに対して新たに形式証明を与えた．具体的には (i) GPUCSL を Frame 規則で拡張した

とき，Blom らの健全性証明の中で用いている補題が成り立たなくなることと，(ii) バリア相違 (barrier divergence) を防ぐための十分条件であるスレッド ID 独立性の前提条件が不十分である点が挙げられる．本研究では (i) を Vafeiadis の証明法 [36] を用いることで，(ii) をスレッド ID 独立性とよく似た性質である，non-interference の証明法を応用することで解決した．

(3) 本研究では複雑なメモリアクセスパターンをもつプログラムを simulating function を用いて検証する手法を提案する．Simulating function は GPGPU プログラムの実行を模倣するようなメタ論理の関数であり，これを用いることでプログラムの計算に関する証明をメタ論理の関数に関する証明に帰着させることができる．本研究では reduce テンプレートに対し simulating function を用いた検証を行い，その有効性を確かめた．また reduce コードテンプレートの検証の中で最適化の余地があることを発見し，最適化されたコードの正しさを検証した．

本研究の貢献は以下の通りである．

- 概略のみが与えられていた Blom らの GPUCSL に形式的な証明を与え，もとの健全性の概略の欠陥を明らかにした．
- データ型やユーザ関数コードでパラメタライズされているコードテンプレートを検証するための枠組みを定義し，検証を支援する Coq ライブラリを作成し，実際に map，reduce テンプレートを検証した
- 複雑なメモリアクセスパターンをもつ GPGPU プログラムを検証する手法として，simulating function を用いるものを提案し，実際に reduce テンプレートの検証で有効に働くことを確かめた
- GPUDSL の 1 つである Accelerate で用いられているテンプレートに最適化の余地を発見し，改良されたテンプレートの検証を行った

本稿は以下の構成からなる．2 章では GPU のための並行分離論理 GPUDSL を提案する．3 章では GPGPU コードを検証するためのライブラリと GPGPU コードを検証するための simulating function を用いた手法について提案する．4 章ではコードテンプレートの検証手法を提案する．5 章では関連研究について述べ，6 章で結論と今後の課題について述べる．

第2章　GPUのための並行分離論理 GPUCSL

本章では GPGPU のための並行分離論理 GPUCSL を提案する．

並行分離論理 (CSL) とは，並行プログラムを検証するための Hoare 論理の 1 つであり，プログラム C，事前条件 P 及び事後条件 Q の 3 つ組 $\{P\}\ C\ \{Q\}$ を導出する規則群から成る．

GPGPU プログラム検証のための CSL の 1 つに，Blom らによる拡張 [10] がある．Blom らは CSL を GPGPU の意味論に拡張し健全性の非形式的な証明の概略を与えた．Blom らの CSL が基づく推論規則は，標準的な CSL とは大きく異なる独自のものである．具体的には，(i) 推論規則に Frame 規則を持たないことと，(ii) メモリ資源に関する仕様と計算結果に関する仕様を分けて記述することなどが挙げられる．Blom らはバリア相違が起こらないことの十分条件としてスレッド ID 独立性を提案したが，実際にそれが十分条件であることの証明は与えていない．

本研究では Blom らの CSL に形式的な証明を与えるために，すでに形式的証明が与えられている Vafeiadis の CSL [36] をもとに，Blom らの CSL を再設計 (GPUCSL) し，その健全性を定理証明支援器 Coq を用いて証明した．健全性の証明は Vafeiadis の健全性の証明を応用した．またスレッド ID 独立性を型システムとして形式化し，その健全性の証明を独自に行った．

また，本研究では GPUCSL の健全性の証明を通して，Blom らの証明の問題点を明らかにした．まず，彼らの CSL の健全性の証明の方針を，Frame 規則をもつ一般の CSL の健全性に適用することは困難である．その理由は，彼らの健全性の証明の中で仮定されている一部の補題が，Frame 規則をもつ CSL では成り立たないからである．また，スレッド ID 独立性の健全性は競合状態が発生しないことを前提として要求するが，これは Blom らの議論では与えられていない．

本章は以下の構成からなる．2.1 節では GPUCSL の対象言語について説明する．2.2 節では GPUCSL の推論規則と健全性について説明する．2.3 節では GPUCSL の健全性の証明の概略を与える．2.4 節では Blom らの研究と本研究の差異について説明する．2.5 節では GPUCSL を用いた簡単なカーネルの検証例を示す．

2.1　言語

GPUCSL は CUDA C [31] のサブセットである WhileB 言語を検証の対象とする．CUDA C は C 言語の拡張であり，CUDA C プログラムは CPU 上で実行されるホストコードと GPU 上で実行されるカーネルからなる．ホストコードはカーネルで用いられるグローバルメモリ領域を GPU 上に確保し，そのアドレスとカーネルを実行するグリッドの構造を指定してカーネルを呼び出す．GPU は指定された構造のグリッドを作成し，呼び出されたカーネルをそのグリッドで実行する．グリッドは複数のスレッドブロックで構成され，スレッドブロックは複数のスレッドで構成される．全てのスレッドブロックは同じ個数のスレッドを持つ．本稿ではグリッド内のブロック数を n_b，各ブロック内のスレッド数を n_t と表記する．また，各スレッドはブロック内で一意な 0 から $n_t - 1$ までの ID を持ち，各ブロックは 0 から $n_b - 1$ までの一意な ID をもつ．これらの ID はカーネル内から変数で参照することができ，WhileB 言語ではそれらの変数をそれぞれ変数 `tid`，`bid` とする．また本稿ではスレッド ID の集合 $\{0, \ldots, n_t - 1\}$ を Tid，ブロック ID の集合 $\{0, \ldots, n_b - 1\}$ を Bid と表記する．各スレッドはスレッド毎に存在するレジスタ，ブロック毎に存在する共有メモリ，全スレッドで共有されるグローバルメモリを用いることができる．レジスタはカーネル中の変数に対応する．共有メモリはカーネルの起動時にカーネル内に記述された共有メモリのための配列宣言によって確保される．共有メモリはスレッドブロック毎に存在し，他のスレッドブロックに属するスレッドから参照されることはない．グローバルメモリは CPU によってのみ確保され

$$x, y, \ldots \in \mathsf{Var}$$

$$E \in \mathsf{Exp} ::= x \mid n \mid E_1 + E_2 \mid E_1 * E_2 \mid \ldots$$

$$B \in \mathsf{BExp} ::= E_1 = E_2 \mid E_1 < E_2 \mid !B \mid B_1 \&\& B_2 \mid \ldots$$

$$L \in \mathsf{LExp} ::= \mathbf{Sh}\ E \mid \mathbf{Gl}\ E \mid L[E]$$

$$C \in \mathsf{Cmd} ::= \mathbf{skip} \mid x := E \mid x := L \mid L := E \mid C_1; C_2 \mid \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\ B\ \mathbf{do}\ C \mid \mathbf{barrier}_b$$

$$P \in \mathsf{Prg} ::= \overline{s[n]}; C$$

図 2.1: WhileB 言語の構文規則

```
1  smem[n_t + 2]; // shared memory allocation
2  t := Gl arr[gid];
3  Sh smem[tid + 1] := t;
4  if (tid == 0) {
5    if (bid == 0) { Sh smem[0] := 0 }
6    else { t := Gl arr[gid - 1]; Sh smem[0] := t } }
7  if (tid == n_t - 1) {
8    if (bid == n_b - 1) { Sh smem[n_t + 1] := 0 }
9    else { t := Gl arr[gid + 1]; Sh smem[n_t + 1] := t } }
10 barrier_0;
11 t0 := Sh smem[tid]; t1 := Sh smem[tid + 1]; t2 := Sh smem[tid + 2];
12 Gl out[gid] = t0 + t1 + t2;
```

図 2.2: `stencil` カーネル

る．また，ブロック内のスレッドはバリア同期によって他のスレッドと同期を取ることができる．バリア同期命令を実行したスレッドはブロック内の他の全スレッドが同じバリア同期命令を実行するまで待機する．本研究では 1 グリッドによるカーネルの実行のみを検証の対象とし，ホストコードの実行は対象としない．

バリア同期は，カーネル上で同じ位置にあるバリア同期命令にブロック内の全てのスレッドが到達した時のみ成功する．2 つのスレッドが異なる 2 つのバリア同期命令に同時に到達したときの状態をバリア相違 (barrier divergence [31]) と呼び，このときの振る舞いは未定義である．カーネルがバリア相違を起こさないことは，GPUCSL が保証することの 1 つである．

本研究では GPU による 1 つのカーネルの実行のみを検証対象とする．カーネルを記述する言語として，While 言語にバリア同期命令とグローバルメモリの読み書きを追加した言語 (WhileB 言語) を定義した．CUDA C の機能のうち，WhileB 言語では省略した機能は 8 節で議論する．

2.1.1　構文規則

図 2.1 は WhileB 言語の構文規則である．WhileB 言語は While 言語に共有メモリ及びグローバルメモリの読み書きとバリア同期のための構文を追加したものである．E，B はそれぞれメモリアクセスを含まない算術式及びブール式である．データ型は整数がある．L はメモリアドレスを表す式であり，グローバルメモリ上のアドレス E を表す $\mathbf{Gl}\ E$，共有メモリ上のアドレス E を表す $\mathbf{Sh}\ E$，及びアドレス L からオフセット E だけ離れたアドレスを表す $L[E]$ からなる．C はコマンドであり，While 言語の構文規則に加えてメモリの読み書き ($x := L$，$L := E$) とバリア同期命令 ($\mathbf{barrier}_b$) を持つ．バリア同期命令はプログラム中で一意な自然数の添字 b を持つ．カーネル P は共有メモリ宣言の列 $\overline{s[n]}$ と 1 つのコマンドからなる．各共有メモリ宣言は変数名 s とその長さ n からなり，変数名 s で参照される長さが n の配列を共有メモリに確保することを表す．

WhileB 言語で記述されたカーネルの例として，`stencil` カーネルを図 2.2 に示す．`stencil` カーネルは 1 次元ステンシル計算を行うカーネルである．1 次元ステンシル計算とは入力配列から出力配列を求める計算で，出力配列の i 番目の要素が入力配列の i 番目の要素の近傍 (ここでは $i-1, i, i+1$ 番目の要

$$
\begin{array}{ll}
v \in \mathsf{Val} := \mathbb{Z} & \\
l \in \mathsf{Loc} := \{\mathbf{Sh}\ v \mid v \in \mathbb{Z}\} \cup \{\mathbf{Gl}\ v \mid v \in \mathbb{Z}\} & ts \in \mathit{ThreadState} := \{(c, s) \mid c \in \mathsf{Cmd}, s \in \mathsf{stack}\} \\
s \in \mathsf{stack} := \mathsf{Var} \to \mathsf{Val} & bs \in \mathit{BlockState} := \{(\overline{ts}, h_s) \mid \overline{ts} \in \mathit{ThreadState}^{n_t}, h_s \in \mathsf{sheap}\} \\
h \in \mathsf{heap} := \mathsf{Loc} \rightharpoonup \mathsf{Val} & gs \in \mathit{GridState} := \{(\overline{bs}, h_g) \mid \overline{bs} \in \mathit{BlockState}^{n_b}, h_g \in \mathsf{sheap}\} \\
sh \in \mathsf{sheap} := \mathsf{Val} \rightharpoonup \mathsf{Val} &
\end{array}
$$

図 2.3: スレッド，スレッドブロック，グリッドの状態

$$
\frac{bs_i = \left(\overline{ts}, h_s\right) \qquad \left(\overline{ts}, \mathbf{Sh}\ h_s \uplus \mathbf{Gl}\ h_g\right) \to_b \left(\overline{ts'}, \mathbf{Sh}\ h'_s \uplus \mathbf{Gl}\ h'_g\right)}{\left(\overline{bs}, h_g\right) \to_g \left(\overline{bs}\left[\left(\overline{ts'}, h'_s\right)/i\right], h'_g\right)} \quad \text{(G-Step)}
$$

$$
\frac{bs_i = \left(\overline{ts}, h_s\right) \qquad \left(\overline{ts}, \mathbf{Sh}\ h_s \uplus \mathbf{Gl}\ h_g\right) \to_b \mathbf{abort}}{\left(\overline{bs}, h_g\right) \to_g \mathbf{abort}} \quad \text{(G-Abort)}
$$

図 2.4: グリッド実行の意味論

$$
\frac{(ts_i, h) \to_T (ts', h')}{(\overline{ts}, h) \to_B (\overline{ts}[ts'/i], h')} \quad \text{(B-Step)} \qquad \frac{\forall i \in \mathsf{Tid}, \mathsf{wait}(ts_i.\mathsf{C}) = (b, C'_i) \wedge ts'_i = (C'_i, ts_i.\mathsf{s})}{(\overline{ts}, h) \to_B (\overline{ts'}, h)} \quad \text{(B-Barrier)}
$$

$$
\frac{(ts_i, h) \to_T \mathbf{abort}}{\left(\overline{ts}, h\right) \to_B \mathbf{abort}} \quad \text{(B-Abort)}
$$

図 2.5: ブロック実行の意味論

素) から決まるような計算である．`stencil` カーネルはグリッド内のスレッド数 $n_t n_b$ と等しい長さの配列 `arr` を受け取って，各要素の近傍の和をとった配列を計算する．スレッド IDi，ブロック IDj をもつスレッド (以下スレッド i，j と呼ぶ) は出力配列の $i + jn_t$ 番目の計算を担当する．プログラム中の `gid` は `tid + bid * ` n_t の略記である．`stencil` は配列 `arr` を n_b 個の長さ n_t の連続な領域に分割し，それぞれの領域を各ブロックの共有メモリにコピーする．このとき，共有メモリの端の要素には隣の領域の端の値，存在しない場合は 0 を書き込む．その後共有メモリを使って近傍の和を求め，出力配列に書き込む．

2.1.2　WhileB 言語の操作的意味論

WhileB 言語の操作的意味論は，Vafeiadis [36] の意味論にグリッド，スレッドブロックといったスレッド階層を加え，スレッドブロックの遷移規則にバリア同期のための遷移規則を定義した．意味論は状態間の 2 項関係で表される．図 2.3 は状態の定義である．グリッド状態 $\mathit{GridState}$ はスレッドブロック状態の列 $\overline{bs}$ とグローバルメモリ h_g からなる．グローバルメモリは sheap で定義され，値 Val から値 Val への部分関数である．スレッドブロック状態 $\mathit{BlockState}$ はスレッド状態の列 ts と共有メモリ h_s からなる．h_s は h_g と同様に sheap である．スレッド状態は現在実行しているコマンド C とスタック s の組である．スタック s は変数から値への関数である．図 2.4，2.5，2.6 が遷移規則の定義である．規則はグリッド実行のための遷移規則 $\to_G$: $\mathit{GridState} \times (\mathit{GridState} \cup \mathbf{abort}) \to \mathsf{Prop}$，スレッドブロック実行のための遷移規則 $\to_B$: $(\mathit{ThreadState}^{n_t} \times \mathsf{heap}) \times (\mathit{ThreadState}^{n_t} \times \mathsf{heap} \cup \{\mathbf{abort}\}) \to \mathsf{Prop}$ と，逐次実行のための遷移規則 $\to_T$: $(\mathit{ThreadState} \times \mathsf{heap}) \times (\mathit{ThreadState} \times \mathsf{heap} \cup \{\mathbf{abort}\}) \to \mathsf{Prop}$ からなる．ここで heap は Loc

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

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

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

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

$$\frac{[\![L]\!](s) = l \qquad h(l) = v'}{(x := [L], s, h) \to_t (\textbf{skip}, s[v'/x], h)}\ \text{(T-Read)} \qquad \frac{[\![L]\!](s) = l \qquad h(l) = v' \qquad [\![E]\!](s) = v}{([L] := E, s, h) \to_t (\textbf{skip}, s, h[v/l])}\ \text{(T-Write)}$$

$$\frac{[\![L]\!](s) = l \qquad h(l) = \bot}{(x := [L], s, h) \to_t \textbf{abort}}\ \text{(T-ReadA)} \qquad \frac{[\![L]\!](s) = l \qquad h(l) = \bot}{([L] := E, s, h) \to_t \textbf{abort}}\ \text{(T-WriteA)}$$

図 2.6: 逐次実行の意味論

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

図 2.7: Definition of the wait function

$[\![-]\!] : \mathsf{Exp} \to \mathsf{stack} \to \mathsf{Val}$

$[\![n]\!](s) = n$

$[\![x]\!](s) = s(x)$

$[\![E_1 + E_2]\!](s) = [\![E_1]\!](s) + [\![E_2]\!](s)$

$[\![E_1 * E_2]\!](s) = [\![E_1]\!](s) + [\![E_2]\!](s)$

$[\![-]\!] : \mathsf{BExp} \to \mathsf{stack} \to \{\textbf{true}, \textbf{false}\}$

$[\![E_1 = E_2]\!](s) = [\![E_1]\!](s) = [\![E_2]\!](s)$

$[\![E_1 < E_2]\!](s) = [\![E_1]\!](s) \leq [\![E_2]\!](s)$

$[\![!B]\!](s) = \neg [\![B]\!](s)$

$[\![B_1 \&\& B_2]\!](s) = [\![B_1]\!](s) \wedge [\![B_2]\!](s)$

$[\![-]\!] : \mathsf{LExp} \to \mathsf{stack} \to \mathsf{Loc}$

$[\![\textbf{Sh } E]\!](s) = \textbf{Sh } [\![E]\!](s)$

$[\![\textbf{Gl } E]\!](s) = \textbf{Gl } [\![E]\!](s)$

$$[\![L[E]]\!](s) = \begin{cases} \textbf{Sh } (E' + [\![E]\!](s)) & ([\![L]\!]\ (s) = \textbf{Sh}\ E') \\ \textbf{Gl } (E' + [\![E]\!](s)) & ([\![L]\!]\ (s) = \textbf{Gl}\ E') \end{cases}$$

図 2.8: 式の意味論

から Val への部分関数であり，Loc は共有メモリのアドレス $\{\textbf{Sh } v \mid v \in \mathbb{Z}\}$ とグローバルメモリのアドレス $\{\textbf{Gl } v \mid v \in \mathbb{Z}\}$ の直和である．$S \to_{\{T,B,G\}} S'$ は S を 1 ステップ実行した結果が S' になることを表し，

$$\frac{\begin{array}{c} P \Rightarrow \bigstar_{i\in\mathsf{Tid}}\, P_i \\ \forall i \in \mathsf{Tid},\ \mathsf{BS}, i \vdash_T \{P_i \wedge \mathtt{tid} = i\}\ C\ \{Q_i\} \\ \bigstar_{i\in\mathsf{Tid}}\, Q_i \Rightarrow Q \\ \forall b,\ \bigstar_{i\in\mathsf{Tid}}\, \mathsf{BS}(i,b)_{pre} \Rightarrow \bigstar_{i\in\mathsf{Tid}}\, \mathsf{BS}(i,b)_{post} \\ \Gamma \vdash C : \tau \\ \forall i, b. P_i, Q_i \text{ and } \mathsf{BS}(i,b) \text{ are thread ID independent} \\ \forall i, b, \mathsf{BS}(i,b)_{pre/post} \text{ is precise} \end{array}}{\Gamma \vdash_B \{P\}\ C\ \{Q\}}\ (\textsc{Block})$$

$$\frac{\begin{array}{c} P \Rightarrow \bigstar_{j\in\mathsf{Bid}}\, P_j \\ \forall j \in \mathsf{Bid},\ \Gamma \vdash_B \{P_b \star [\![(\overline{s[n]})]\!] \wedge \mathtt{bid} = j\}\ C\ \{Q_b \star [\![\overline{s[n]}]\!]\} \\ \bigstar_{j\in Bid}\, Q_j \Rightarrow Q \\ \mathsf{disjoint}(\overline{s}) \qquad \overline{s} \cap \mathsf{writes}(C) = \emptyset \\ \forall b. \mathtt{tid}, \mathtt{bid} \notin P_b \wedge \mathrm{fv}(Q_b) = \emptyset \qquad \mathtt{tid}, \mathtt{bid} \notin \overline{s} \\ E \vdash \overline{s} : \mathsf{Lo} \qquad E \vdash \mathtt{tid} : \mathsf{Hi} \qquad E \vdash \mathtt{bid} : \mathsf{Lo} \end{array}}{\vdash_G \{P\}\ \overline{s[n]}; C\ \{Q\}}\ (\textsc{Grid})$$

図 2.9: 並列実行のための証明規則

$S' = \mathbf{abort}$ のときは S を 1 ステップ動作させた結果がエラーになることを表す．図 2.4 がグリッド実行の意味論 ($\rightarrow_g$) の定義である．スレッドブロックのうちの 1 つを選択し，そのスレッドブロックの共有メモリとグリッドのグローバルメモリの和 ($\mathbf{Sh}\ h_s \uplus \mathbf{Gl}\ h_g$) をとり，そのヒープの上でスレッドブロックの意味論 $\rightarrow_B$ で動作されることを意味する．共有メモリ h_s とグローバルメモリ h_g の和 $\mathbf{Sh}\ h_s \uplus \mathbf{Gl}\ h_g$ は，アドレスが共有メモリのもの ($\mathbf{Sh}\ v$) の時は $h_s(v)$ を返し，グローバルメモリのもの ($\mathbf{Gl}\ v$) のときは $h_g(v)$ を返す heap である．図 2.5 がスレッドブロックの意味論 $\rightarrow_B$ の定義である．ブロック実行はグリッド実行と同様にブロック内の 1 つのスレッドを選択して動作させるか (B-Step, B-Abort)，バリア同期を実行する (B-Barrier)．バリア同期はスレッドブロック内の全てのスレッドがあるバリア b に到達したときのみ成功する．前提の $ts_i.\mathsf{C}$ 及び $ts_i.\mathsf{s}$ はスレッド状態 ts_i のコマンド及びスタックを表す．関数 $\mathsf{wait}(C)$ は次に実行する命令がバリア同期のとき定義され，その ID とバリア同期を取り除いたコマンドを返す (図 2.7)．

図 2.6 が逐次実行の意味論 ($\rightarrow_T$) の定義である．それぞれ通常の While 言語と同様に定義される．$[\![-]\!]$ は式の表示的意味論であり，図 2.8 で定義される．

2.2 GPUCSL

CSL は並行プログラムが仕様 $\{P\}\ C\ \{Q\}$ を満たすことを検証するための論理である． 直感的には仕様 $\{P\}\ C\ \{Q\}$ は，事前条件を満たすメモリ状態のもとでコマンド C を開始した時，(i) 実行中にアボートせず， (ii) 停止するとき，その状態で Q が成り立つことを表す．ここで，P と Q はスタックとヒープの組についての述語 (言明) である．

CSL は結論 $\{P\}\ C\ \{Q\}$ を証明するための規則群からなる．CSL の並行実行のための証明規則は,「各スレッドがアクセスするメモリ領域が互いに独立ならば，実行後の状態はそれらの独立な領域を合わせたものになる」という直感に基づく．つまり，各スレッドがプログラム C_1，C_2 を P_1，P_2 で表現されるメモリ領域のもとで安全に実行し，停止後の状態が Q_1，Q_2 を導くことが証明できるならば，プログラム全体として仕様 $\{P_1 \star P_2\}\ C_1 \| C_2\ \{Q_1 \star Q_2\}$ が成り立つことである．ここで $C_1 \| C_2$ は並列実行を表し，$P_1 \star P_2$ (分離積，separating conjunction) はメモリが独立な 2 つの領域に分割できて，それぞれの領域が P_1，P_2 を満たすことを示す．

$$v \in \mathsf{Val} \coloneqq \mathbb{Z}$$

$$l \in \mathsf{Loc} \coloneqq \{\mathbf{Sh}\ v \mid v \in \mathbb{Z}\} \cup \{\mathbf{Gl}\ v \mid v \in \mathbb{Z}\}$$

$$s \in \mathsf{stack} \coloneqq \mathsf{Var} \to \mathsf{Val}$$

$$h \in \mathsf{pheap} \coloneqq \mathsf{Loc} \rightharpoonup \mathsf{Perm} \times \mathsf{Val}$$

$$(p_1, v_1) \perp (p_2, v_2) \overset{\triangle}{\Leftrightarrow} p_1 + p_2 \leq 1 \wedge v_1 = v_2$$

$$(p_1, v_1) \oplus (p_2, v_2) \overset{\triangle}{\Leftrightarrow} (p_1 + p_2, v_1)$$

$$h_1 \perp h_2 \overset{\triangle}{\Leftrightarrow} \forall l.h_1(l) = \perp \vee h_2(l) = \perp \vee h_1(l) \perp h_2(l)$$

$$(h_1 \uplus h_2)(l) \coloneqq \begin{cases} h_1(l) \oplus h_2(l) & (h_1(l), h_2(l) \neq \perp) \\ h_1(l) & (h_2(l) = \perp) \\ h_2(l) & (h_1(l) = \perp) \end{cases}$$

$$s, h \vDash \mathbf{emp} \overset{\triangle}{\Leftrightarrow} \forall l.h(l) = \perp$$

$$s, h \vDash L \mapsto^p E \overset{\triangle}{\Leftrightarrow} \forall l.h(l) = \begin{cases} (p, [\![E]\!](s)) & (l = [\![L]\!](s)) \\ \perp & (\text{otherwise}) \end{cases}$$

$$s, h \vDash P \star Q \overset{\triangle}{\Leftrightarrow} \exists h_1 h_2.h_1 \perp h_2 \wedge h = h_1 \uplus h_2 \wedge s, h_1 \vDash P \wedge s, h_2 \vDash Q$$

$$s, h \vDash P \wedge Q \overset{\triangle}{\Leftrightarrow} s, h \vDash P \wedge s, h \vDash Q$$

$$s, h \vDash \exists v.P \overset{\triangle}{\Leftrightarrow} \exists v.(s, h \vDash P)$$

$$s, h \vDash E_1 = E_2 \overset{\triangle}{\Leftrightarrow} [\![E_1]\!](s) = [\![E_2]\!](s)$$

$$L \mapsto^p - \coloneqq \exists v.L \mapsto^p v$$

$$\mathsf{array}_p(L, m, n, f) \coloneqq \bigstar_{i=m}^{m+n-1} (L[i] \mapsto^p f(i))$$

$$\mathsf{sarray}_p(L, m, n, f, d, i) \coloneqq \bigstar_{j=m, j \bmod d = i}^{m+n-1} (L[j] \mapsto^p f(j))$$

$$\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R \coloneqq (P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$$

図 2.10: GPUCSL の論理式の意味論

GPUCSL では Grid 規則，Block 規則 (図 2.9) を用いてグリッド，スレッドブロックの並列実行を検証する．グリッド実行，スレッドブロック実行を検証するためには，共通して以下のことを示す必要がある (それぞれの条件が前提の何行目に対応するかを示している).

1 行目　仕様の事前条件が各ブロック (スレッド) の事前条件の分離積を導くこと

2 行目　任意のスレッドに対して，そのスレッドが逐次実行の意味論で仕様 $\{P_i \wedge \ldots\}\ C\ \{Q_i\}$ を満たすこと

3 行目　各スレッドの事後条件の分離積が仕様の事後条件を導くこと

またブロック実行に関してはバリア同期が安全に行われることを示すために，以下のことを示す必要がある．

4 行目　バリア仕様 *BS* が，バリア同期の前後で過不足なくリソースの再配分を行うものになっていること．これは，各スレッドの担当するメモリ領域が，バリア同期の前後で正しく交換されること意味する．

5，6 行目　カーネルがスレッド ID 独立性を満たすこと．直感的には，スレッド ID に依存しない実行文脈の下にのみバリア同期命令が現れることを意味する．

本節ではまず GPUCSL の言明の意味論について述べる．次に GPUCSL の証明規則について述べる．次にスレッド ID 独立性について述べ，最後に 3 つ組の意味論について述べる．

2.2.1　言明の意味論

GPUCSL の言明は通常の述語論理の論理式に加えて，ヒープのための言明 **emp**，$L \mapsto^p E$ 及び $P \star Q$ 及びそれらを用いて定義される言明からなる．言明はスタック s とパーミッション付きヒープ h のもとで評価される．図 2.10 は GPUCSL の言明の定義である．パーミッション付きヒープ pheap はアドレスからパーミッションと値の組への部分関数 $\mathsf{Loc} \rightharpoonup \mathsf{Perm} \times \mathsf{Val}$ である．パーミッションはそのヒープを所有するスレッドがそのアドレスに対して行うことができる操作を意味し，本研究では有理数パーミッ

ション [11] (Perm := $\{q \in \mathbb{Q} \mid 0 < q \leq 1\}$) を用いる．スレッドがアドレス l に対してパーミッション 1 を持つとき，そのスレッドはそのアドレスに対して書き込みと読み込みの両方を行うことができ，1 より小さいパーミッションを持つときは読み込みのみを行うことができる．

パーミッション付きヒープ h_1, h_2 に対して，その和 $h_1 \uplus h_2$ は h_1, h_2 が独立 ($h_1 \perp h_1$) なときに限って定義され，各アドレスに対してパーミッションの和をとったヒープとして定義される．$h_1 \perp h_2$ は全てのアドレス l に対して

- $h_1(l)$ か $h_2(l)$ が定義されない，または
- $h_1(l)$ か $h_2(l)$ の両方が定義され，かつその値が等しくパーミッションの総和が 1 を超えない ($h_1(l) \perp h_2(l)$)

ことと定義される．

emp は h がどのアドレスでも定義されないときに真になる．$L \mapsto^p E$ は h が L の評価結果のみで定義され，その値が E でパーミッションが p であるときに真となる．$P \star Q$ は h が独立なヒープ h_1, h_2 の和として表され，それぞれのヒープで P, Q が成り立つときに真となる．GPUCSL の言明は通常のパーミッション付き CSL に加えて，配列を表す言明として $\mathsf{array}_p(L, m, n, f)$，及び $\mathsf{sarray}_p(L, m, n, f, d, i)$ をもつ．$\mathsf{array}_p(L, m, n, f)$ は $L[m]$ が長さ n の配列で，各 $m \leq i < m + n - 1$ に対してアドレス $L[i]$ が値 $f(i)$ を持つことを表す．p はパーミッションで，配列中の全ての位置に対して同じパーミッション p をもつ．$\mathsf{sarray}_p(L, s, n, f, d, i)$ は配列 L の m から $m + n - 1$ までの位置のうち，d で割った剰余が i となるような位置についてパーミッション p を持つことを表す．各アドレス $L[j]$ 番目の要素は $\mathsf{array}_p(L, m, n, f)$ と同様に値 $f(i)$ をもつ．$\mathsf{sarray}_p(L, m, n, f, d, i)$ は d 個のスレッドが配列の d 個ずつ離れた位置にアクセスするようなカーネルを検証する際に用いる．$\mathsf{array}_p(L, m, n, f)$，$\mathsf{sarray}_p(L, m, n, f, d, i)$ に関して以下の性質が成り立つ．

- $\bigstar_{i=0}^{d-1} \mathsf{array}_{1/d}(L, m, n, f) \Leftrightarrow \mathsf{array}_1(L, m, n, f)$ (array_distribute)
- $\bigstar_{i=0}^{d-1} \mathsf{sarray}_p(L, m, n, f, d, i) \Leftrightarrow \mathsf{array}_p(L, m, n, f)$ (sarray_distribute)
- $0 \leq j < n \Rightarrow (\mathsf{array}_p(L, m, n, f) \Leftrightarrow$
  $\mathsf{array}_p(L, m, j, f) \star L[m + j] \mapsto^p f(m + j) \star \mathsf{array}_p(L, m + j + 1, n - j - 1, f))$ (array_forward)
- $0 \leq j < n \wedge (m + j) \bmod d = i \Rightarrow$
  $(\mathsf{sarray}_p(L, m, n, f, d, i) \Leftrightarrow$
  $\mathsf{sarray}_p(L, m, j, f, d, i) \star L[m + j] \mapsto^p f(m + j) \star \mathsf{sarray}_p(L, m + j + 1, n - j - 1, f, d, i))$ (sarray_forward)
- $(\forall 0 \leq j < n, (m + j) \bmod d = i \Rightarrow f_1(m + j) = f_2(m + j)) \Rightarrow$
  $\mathsf{sarray}_p(L, m, n, f_1, d, i) \Leftrightarrow \mathsf{sarray}_p(L, m, n, f_2, d, i)$ (sarray_eq)

これらの性質は配列の長さ n に関する帰納法で容易に証明できる．これらの性質は配列に並列にアクセスするカーネルの検証に用いる．array_distribute は配列に対する読み込み専用のパーミッションを，各スレッドに配布できることを表す．sarray_distribute は d で割った剰余が i となる添字のパーミッションを各スレッドに配布できることを表す．array_forward，sarray_forward は $\mathsf{array}(L, m, n, f)$ や $\mathsf{sarray}(L, m, n, f, d, i)$ が成り立つとき，配列内の 1 つの位置に関するパーミッションを取り出せることを表す．

2.2.2　証明規則

GPUCSL は 3 種類の結論 $\mathsf{BS}, i \vdash_T \{P\}\ C\ \{Q\}$，$\Gamma \vdash_B \{P\}\ C\ \{Q\}$ 及び $\vdash_G \{P\}\ \overline{s[n]}; C\ \{Q\}$ を導出する規則群からなる．それぞれ各スレッドの逐次実行，各スレッドブロックの実行，グリッドの実行に対応する．図 2.11 は逐次実行に関する証明規則である．Barrier 規則以外は通常のパーミッション付き CSL と同様である．Frame 規則は，証明した仕様 $\{P\}\ C\ \{Q\}$ をカーネル C が言及しないリソース R で拡張する．前提に現れる $\mathrm{fv}(R)$ と $\mathrm{wr}(C)$ はそれぞれ R に現れる局所変数の集合と，C に現れる変数の中で代入文の左辺に出現する変数の集合を表す．Barrier 規則はブロック内のスレッドがバリア仕様 BS で定められたメモリ資源の交換を行うことを表す．$\mathsf{BS}_{pre}(i, b), \mathsf{BS}_{post}(i, b)$ はそれぞれバリア b を実行する前に i 番目のスレッドが返却する資源と，実行後にスレッド i に配布される資源を表す．BS は副条件としてバリア同期の前後で交換される資源の総和が保存されることが要求される．つまり任意の b に対して，

$$\frac{}{\mathsf{BS}, i \vdash_T \{Q\}\ \mathbf{skip}\ \{Q\}}\ (\textsc{Skip}) \qquad \frac{}{\mathsf{BS}, i \vdash_T \{P[E/x]\}\ x := E\ \{P\}}\ (\textsc{Assign})$$

$$\frac{x \notin \mathrm{fv}(L) \quad x \notin \mathrm{fv}(E)}{\mathsf{BS}, i \vdash_T \{L \mapsto^{\pi} E\}\ x := L\ \{L \mapsto^{\pi} E \wedge x = E\}}\ (\textsc{Read})$$

$$\frac{}{\mathsf{BS}, i \vdash_T \{L \mapsto^{1} E_1\}\ L := E_2\ \{L \mapsto^{1} E_2\}}\ (\textsc{Write})$$

$$\frac{\mathsf{BS}, i \vdash_T \{P\}\ C\ \{Q\} \quad \mathrm{fv}(R) \cap \mathsf{writes}(C) = \emptyset}{\mathsf{BS}, i \vdash_T \{P \star R\}\ C\ \{Q \star R\}}\ (\textsc{Frame}) \qquad \frac{\mathsf{BS}, i \vdash_T \{P\}\ C_1\ \{Q\} \quad \mathsf{BS}, i \vdash_T \{Q\}\ C_2\ \{R\}}{\mathsf{BS}, i \vdash_T \{P\}\ C_1; C_2\ \{R\}}\ (\textsc{Seq})$$

$$\frac{\mathsf{BS}, i \vdash_T \{P \wedge B\}\ C_1\ \{Q\} \quad \mathsf{BS}, i \vdash_T \{P \wedge \neg B\}\ C_2\ \{Q\}}{\mathsf{BS}, i \vdash_T \{P\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \{Q\}}\ (\textsc{If}) \qquad \frac{\mathsf{BS}, i \vdash_T \{P \wedge B\}\ C\ \{P\}}{\mathsf{BS}, i \vdash_T \{P\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \{P \wedge \neg B\}}\ (\textsc{While})$$

$$\frac{}{\mathsf{BS}, i \vdash_T \{\mathsf{BS}(i,b)_{pre}\}\ \mathbf{barrier}_b\ \{\mathsf{BS}(i,b)_{post}\}}\ (\textsc{Barrier}) \qquad \frac{P \Rightarrow P' \quad \mathsf{BS}, i \vdash_T \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\mathsf{BS}, i \vdash_T \{P\}\ C\ \{Q\}}\ (\textsc{Conseq})$$

図 2.11: GPUCSL の証明規則

$$\frac{}{\Gamma \vdash \mathtt{tid} : \mathsf{Hi}}\ (\textsc{Ty-TID}) \qquad \frac{}{\Gamma \vdash n : \mathsf{Lo}}\ (\textsc{Ty-Const})$$

$$\frac{x \neq \mathtt{tid}}{\Gamma \vdash x : \Gamma(x)}\ (\textsc{Ty-Var}) \qquad \frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \oplus E_2 : \tau_1 \sqcup \tau_2}\ (\textsc{Ty-Op})$$

図 2.12: 式に対する型付け規則

$\star_{i \in \mathsf{Tid}} BS_{pre}(i,b) \Rightarrow \star_{i \in \mathsf{Tid}} BS_{post}(i,b)$ が成り立つことである．この条件はスレッドブロック実行のための証明規則の前提に出現する．

図 2.9 はスレッドブロック実行のための証明規則 (Block 規則) とグリッド実行のための証明規則 (Grid 規則) である．Block 規則は各スレッド毎に独立な資源 P_i，Q_i のもとで検証できるならばそれらの資源を 1 つに集約した資源 P，Q のもとでブロック実行が正しく行われることを意味する (前提の 1 行目から 3 行目)．前提の 4 行目はバリア仕様がバリア同期の前後で資源を保存することを検証する．5 行目，6 行目の条件はバリア相違 (barrier divergence) [31] を起こさないことを保証するための規則であり，2.2.3 節で説明する．7 行目はバリア仕様が precise であることを表す．ここで言明 P が precise であるとは，任意の h について，$s, h_1 \models P$ を満たす h の部分ヒープ h_1 が 1 つしか存在しないこと，つまり $\forall s, h, h_1, h_1', h_2, h_2'. h = h_1 \uplus h_1' \wedge h = h_2 \uplus h_2' \wedge s, h_1 \models P \wedge s, h_2 \models P \Rightarrow h_1 = h_2$ が成り立つことと定義される．これは Reynolds の反例 [33] を排除するための規則である．

Grid 規則は Block 規則と同様に各ブロックに資源を分配し，各ブロックごとに検証を行う (前提の 1 行目から 3 行目)．事前条件と事後条件に現れる $[\![\overline{s[n]}]\!]$ は共有メモリに関する条件を表し，$[\![\overline{s[n]}]\!] = \star_{s_i[n_i] \in \overline{s[n]}}\ \mathsf{array}(\mathbf{Sh}\ s_i, n_i, -)$ と定義される．その他の条件については健全性の証明の簡略化のために導入した条件であり，本質的ではないため説明を省略する．

$$\frac{\Gamma \vdash E : \tau}{\Gamma \vdash \mathbf{Sh}\ E : \tau}\ (\textsc{Ty-SLoc}) \qquad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash \mathbf{Gl}\ E : \tau}\ (\textsc{Ty-SLoc})$$

$$\frac{\Gamma \vdash L : \tau_1 \qquad \Gamma \vdash E : \tau_2}{\Gamma \vdash L[E] : \tau_1 \sqcup \tau_2}\ (\textsc{Ty-Var})$$

図 2.13: LExp に対する型付け規則

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 = E_2 : \tau_1 \sqcup \tau_2}\ (\textsc{Ty-Eq}) \qquad \frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 \leq E_2 : \tau_1 \sqcup \tau_2}\ (\textsc{Ty-Ineq})$$

$$\frac{\Gamma \vdash B_1 : \tau_1 \quad \Gamma \vdash B_2 : \tau_2}{\Gamma \vdash B_1 \&\& B_2 : \tau_1 \sqcup \tau_2}\ (\textsc{Ty-And}) \qquad \frac{\Gamma \vdash B : \tau}{\Gamma \vdash !B : \tau}\ (\textsc{Ty-Not})$$

図 2.14: ブール式に対する型付け規則

$$\frac{}{\Gamma \vdash \mathbf{skip} : \tau}\ (\textsc{Ty-skip}) \qquad \frac{\Gamma \vdash L : \tau \quad \tau \sqcup \tau' \sqsubseteq \Gamma(x)}{\Gamma \vdash x := L : \tau'}\ (\textsc{Ty-Read})$$

$$\frac{}{\Gamma \vdash L := E : \tau}\ (\textsc{Ty-Write}) \qquad \frac{\Gamma \vdash E : \tau \quad \tau \sqcup \tau' \sqsubseteq \Gamma(x)}{\Gamma \vdash x := E : \tau'}\ (\textsc{Ty-Assign})$$

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

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

図 2.15: コマンドのための型付け規則

2.2.3　スレッド ID 独立性

Blom らはバリア相違が起こらないことを保証する条件としてスレッド ID 独立性を提案した [10]．カーネル中の命令や変数がスレッド ID 独立であるとは，その実行や値がスレッド ID 値の影響を直接，間接に受けないことを意味する．したがって，スレッド ID 独立な命令の実行トレースは全スレッドで一致する．バリア同期命令がスレッド ID 独立であれば，バリア相違は起こらないことが保証される．スレッド ID 独立性の定義は健全ではあるが，完全ではない [10]．そのため，バリア相違を起こさないがスレッド ID 独立性を満たさないような，GPUCSL で検証することができないプログラムがある．

我々はスレッド ID 独立性を型システムとして再形式化した (図 2.12，2.13，2.14，2.15)．型システムは文献 [30] の non-interference を参考に設計した．スレッド ID 独立性では変数，式，ブール式，アドレス式及びコマンドに Hi，Lo のいずれかの型を付ける．型 Lo はスレッド ID 独立であることを意味する．Γ は型付け環境で，$\Gamma : \mathsf{Var} \to \{\mathsf{Hi}, \mathsf{Lo}\}$ である．図 2.12 は式についての型付け規則である．式の型はその式が型 Hi をもつ変数を含むとき Hi，それ以外の時は Lo とする．$\tau_1 \sqcup \tau_2$ は以下のように定義される．

$$\tau_1 \sqcup \tau_2 \quad = \quad \begin{cases} \mathsf{Lo} & (\tau_1 = \mathsf{Lo} \wedge \tau_2 = \mathsf{Lo}\ \text{のとき}) \\ \mathsf{Hi} & (\text{それ以外のとき}) \end{cases}$$

LExp やブール式に対する型付け規則も同様に定義される (図 2.13，図 2.14)．図 2.15 がコマンドのための型付け規則である．コマンド C に型 Lo が付くとき，そのコマンドはスレッド ID に依存しない文脈で実行されることが保証される．$\tau_1 \sqsubseteq \tau_2$ は $\tau_1 = \mathsf{Lo} \vee \tau_2 = \mathsf{Hi}$ と定義する．規則 Ty-Read，Ty-Assign は型 Lo をもつ変数の値が，型 Hi をもつ変数の値に依存することを禁止する．規則 Ty-While，Ty-If は，型 Hi をもつ変数の値に依存する条件式のもとに，型 Lo をもつコマンドが出現することを禁止する．バリア同期命令には型 Lo を付けることで，バリア同期に必ず成功することを保証することができる (規則 Ty-Barrier)．言明 P がスレッド ID 独立 (thread ID independent) であるとは，P が $\Gamma(x) = \mathsf{Hi}$ となる変数 x を含まないことと定義される．

2.2.4　健全性

GPUCSL の健全性の定義は Vafeiadis [36] を参考にした．主な変更点は (i) スレッド階層を反映したことと，(ii) バリア同期のための条件を追加したことである．GPUCSL の健全性を定義するために，述語 $\mathsf{Gsafe}_n(gs, Q), \mathsf{Bsafe}_n(\overline{ts}, h, Q, \Gamma), \mathsf{Tsafe}_n(C, s, h, Q, \mathsf{BS})$ を定義する．これらのそれぞれグリッド実行，スレッドブロック実行，スレッド実行に対応し，「少なくとも n ステップ以内の実行は安全」であることを表す．ここで安全とは (i) 停止しているならばそのメモリ状態で事後条件を満たすこと，(ii) **abort** しないこと，(iii) バリア相違を起こさないことを表す．健全性の定義のために，以下の記法を用いる．

定義 1. $gs = (\overline{bs}, h_g)$，$bs_j = (\overline{(C_{i,j}, s_{i,j})}_{i \in \mathsf{Tid}}, h_s(j))$ のとき，$gs.bs = \overline{bs}$，$gs.h_g = h_g$，$gs.C(i,j) = C_{i,j}$，$gs.s(i,j) = s_{i,j}$，$gs.h_s(j) = h_s(j)$ とかく．また，$gs[C(i,j) := C']$ で，$gs.C(i,j)$ を C' で置き換えた C を表す．$gs.[bs := bs']$，$gs[h_g := h']$，$gs.[s(i,j) := s']$，$gs.[h_s(j) := h'_s]$ についても同様に定義される．

バリア相違について議論するために，以下の定義を用いる．

定義 2. *GridState* gs がバリア相違状態にあるとは以下のように定義される．ある $j \in \mathsf{Bid}$ が存在して，

- ある $i_1, i_2 \in \mathsf{Tid}$ が存在して，$\mathsf{wait}(gs.C(i_1, j)) = (-, -)$ かつ $gs.C(i_2, j) = \mathbf{skip}$，または
- ある $i_1, i_2 \in \mathsf{Tid}$ が存在して，$\mathsf{wait}(gs.C(i_1, j)) = (b_1, C'_1)$ かつ $\mathsf{wait}(gs.C(i_2, j)) = (b_2, C'_2)$ かつ $b_1 \neq b_2$

$\mathsf{Gsafe}(gs, Q)$ は以下のように定義される．

定義 3. $\mathsf{Gsafe}_n(gs, Q)$ を以下のように定義する．
$\mathsf{Gsafe}_n(gs, Q)$ は常に成り立つ．
$\mathsf{Gsafe}_{n+1}(gs, Q)$ は以下が成り立つとき，かつそのときに限って成立する．

1. $\forall i \in \mathsf{Tid}, j \in \mathsf{Bid}.\ gs.C(i,j) = \mathbf{skip} \Rightarrow h \vDash Q$
2. 任意の spheap h_F に対して，$gs.h_g \perp h_F$ ならば $gs[h_g := h_g \uplus h_F] \not\rightarrow_G \mathbf{abort}$
3. 任意の $j \in \mathsf{Bid}$ に対して，$gs.bs_j$ はバリア相違を起こしていない
4. 任意の spheap h_F に対して，$gs.h_g \perp h_F$ かつ $gs[h_g := h_g \uplus h_F] \rightarrow_G gs'$ ならばある spheap h'' が存在して $gs'.h_g = h'' \uplus h_F$ かつ $\mathsf{Gsafe}_n(gs'[h_g := h''], Q)$

1 は停止した時の状態で Q が成り立つこと，2 は 1 ステップ以内にアボートしないこと，3 は各ブロックがバリア相違を起こさないこと，4 は 1 ステップ進んだ先の状態で 1 つ小さい Gsafe 述語が成り立つことを意味する．

Gsafe 述語を用いると，$\{P\}\ \overline{s[m]}; C\ \{Q\}$ の意味を以下のように定義できる．

定義 4 ($\vDash \{P\}\ \overline{s[m]}; C\ \{Q\}$ の定義)**.** $\vDash \{P\}\ \overline{s[m]}; C\ \{Q\}$ は以下が成り立つとき，かつそのときに限って成り立つ．任意の n, s について，

1. 任意の $i \in Tid, j \in \mathsf{Bid}$ について，$gs.C(i,j) = C$，かつ
2. 任意の $i \in Tid, j \in \mathsf{Bid}$ について，$gs.s(i,j), gs.h_s(j) \vDash [\![\overline{s[m]}]\!]$，かつ
3. 任意の $i \in Tid, j \in \mathsf{Bid}$ について，$gs.s(i,j)(\mathtt{tid}) = i \wedge gs.s(i,j)(\mathtt{bid}) = j$，かつ
4. 任意の $i, i' \in Tid, j, j' \in \mathsf{Bid}$ について，$v \neq \mathtt{tid}$ かつ $v \neq \mathtt{bid}$ ならば $gs.s(i,j)(v) = s(v)$

5. $s, gs.h_g \models P$

ならば任意の n に関して $\mathsf{Gsafe}_n(gs, Q)$.

1 は各スレッドのコマンドが C であること，2 は各ブロックの共有メモリが共有メモリ宣言に従って初期化されていること，3 は各スレッドの変数 `tid`，`bid` が正しく初期化されていること，4 は各スレッドのスタックが変数 `tid`，`bid` を除いて等しく初期化されていること，5 は初期状態が P を満たすことを表す.

Grid 規則の健全性の証明のために，ブロック実行，逐次実行のための規則についての健全性を定義する．まず Gsafe 述語と同様に述語 $\mathsf{Bsafe}(\overline{ts}, h, Q, \Gamma)$，$\mathsf{Tsafe}(C, s, h, Q, BS)$ 定義し，上と同様の方法で逐次実行についての $\{P\}\ C\ \{Q\}$ の意味を定義する.

定義 5 (Bsafe 述語)**.** $\mathsf{Bsafe}_n(\overline{ts}, h, Q, \Gamma)$ を以下のように定義する.
$\mathsf{Bsafe}_0(\overline{ts}, h, Q, \Gamma)$ は常に成り立つ.
$\mathsf{Bsafe}_{n+1}(\overline{ts}, h, Q, \Gamma)$ は以下が成り立つとき，かつそのときに限って成立する.

1. $\forall i \in Tid.\ ts_i.C = \mathbf{skip}$ ならば $\overline{s}, h \models Q$
2. 任意の pheap h_F に対して，$h \perp h_F$ ならば $(\overline{ts}, h \uplus h_F) \nrightarrow_G \mathbf{abort}$
3. $\forall i \in \mathsf{Tid}.\mathsf{wait}(ts_i.C) = (b_i, C_i')$ ならば $\forall i, j \in \mathsf{Tid}, \Gamma \models ts_i.s =_L ts_j.s$ かつ $b_i = b_j$
4. 任意の pheap h_F に対して，$h \perp h_F$ かつ $(\overline{ts}, h \uplus h_F) \rightarrow_g (\overline{ts'}, h')$ ならばある pheap h'' が存在して $h' = h'' \uplus h_F$ かつ $\mathsf{Bsafe}_n(\overline{ts'}, h'', Q, \Gamma)$

ここで $\Gamma \models s =_L s'$ は $\forall x, \Gamma(x) = \mathsf{Lo} \Rightarrow s(x) = s'(x)$ と定義される．また $\overline{s}, h \models Q$ は $\exists s', (\forall i \in \mathsf{Tid}, \Gamma \models s_i =_L s') \wedge s', h \models Q$ と定義される．Bsafe 述語を用いると，$\{P\}\ C\ \{Q\}$ の意味を以下のように定義できる.

定義 6. $\Gamma \models_B \{P\}\ C\ \{Q\}$ は以下が成り立つとき，かつそのときに限って成り立つ．任意の n，$\overline{ts}$，h について，$\overline{s}, h \models P$ かつ $\forall i, ts_i.s(\mathtt{tid}) = i$ かつ $\forall i. ts_i.C = C$ ならば $\mathsf{Bsafe}_n(\overline{ts}, h, Q, \Gamma)$ が成り立つ.

定義 7 (Tsafe 述語)**.** $\mathsf{Tsafe}_{i,n}(ts, h, Q, BS)$ を以下のように定義する.
$\mathsf{Tsafe}_{i,0}(ts, h, Q, BS)$ は常に成り立つ.
$\mathsf{Tsafe}_{i,n+1}(ts, h, Q, BS)$ は以下が成り立つとき，かつそのときに限って成立する.

1. $ts.C = \mathbf{skip} \Rightarrow ts.s, h \models Q$
2. 任意の pheap h_F に対して，$h \perp h_F$ ならば $(ts, h \uplus h_F) \nrightarrow_T \mathbf{abort}$
3. $\mathsf{writes}(ts) \neq \perp \rightarrow \exists v, h(\mathsf{writes}(ts)) = (1, v)$
4. 任意の $h \perp h_F$ を満たす pheap h_F に対して，$(ts, h \uplus h_F) \rightarrow_T (ts', h')$ ならば，ある pheap h'' が存在して $h' = h'' \uplus h_F$ かつ $\mathsf{Tsafe}_{i,n}(ts', h'', Q, BS)$
5. $\mathsf{wait}(ts.C) = (b, C')$ ならばある pheap h_P, h_F が存在して以下が成り立つ.
   - $h_P \perp h_F$ かつ $h = h_P \uplus h_F$ かつ $ts.s, h_P \models BS(i, b)_{pre}$
   - 任意の $h_Q \perp h_F$ 及び $s, h_Q \models BS(i, b)_{post}$ を満たす pheap h_Q に対して $\mathsf{Tsafe}_{i,n}((C', ts.s), h_Q \uplus h_F, Q, BS)$

(3) で用いられる補助関数 writes はコマンドが次の 1 ステップの実行で書き込み命令を実行するならばその書き込みを行うアドレスを返す補助関数である.

$$\mathsf{writes}((L := E, s)) = [\![L]\!](s)$$
$$\mathsf{writes}((C_1; C_2, s)) = \mathsf{writes}(C_1, s)$$
$$\mathsf{writes}((C, s)) = \perp (\text{その他の場合})$$

(5) は次に実行するコマンドがバリア同期のときのための定義である．これはバリアに到達するとき，各スレッドがバリア仕様にしたがって適切なリソースを返却し，配分されたリソースの下で安全であることを表す.

Tsafe 述語を用いると，逐次実行についての $\{P\}\ C\ \{Q\}$ の意味は以下のように定義できる.

定義 8. $BS, i \models_T \{P\}\ C\ \{Q\}$ は以下のように定義する．任意の自然数 n，スタック s，pheap h に対して，$s, h \models P$ ならば $\mathsf{Tsafe}_{i,n}((C, s), h, Q, BS)$

2.3　健全性証明

本研究では GPUCSL の健全性を Coq を用いて証明した．証明は Vafeiadis の Coq による CSL の健全性証明 [37] を参考に行い，5500 行程度になった．Vafeiadis の Coq による証明は，GPUCSL と異なる点が多いため，本研究では Vafeiadis の証明を再利用せずに一から形式化を行った．

Vafeiadis の体系は同期モデルに atomic 命令を用いているため，我々はバリア同期に適用できるように健全性の定義を変更した．逐次実行についての推論規則の健全性は Vafeiadis の証明 [36] を応用した．スレッド ID 独立性の健全性は独自に証明を与えた．並列実行についての健全性の証明は Vafeiadis の証明の応用して行ったが，スレッド ID 独立性の健全性を用いる点が異なる．

以下では証明の概略を示す．健全性とは以下の定理である．

定理 9 (健全性)**.** $\vdash_G \{P\}\ C\ \{Q\}$ ならば $\models_G \{P\}\ C\ \{Q\}$

定理 9 の証明のために，以下の Block 規則の健全性を示す．

補題 10 (ブロック規則の健全性)**.** $\Gamma \vdash_B \{P\}\ C\ \{Q\}$ ならば $\Gamma \models_B \{P\}\ C\ \{Q\}$

補題 10 の証明のために，まず逐次実行のための推論規則の健全性を補題として証明する．次にスレッド ID 独立性の健全性 (バリア相違を起こさないこと，バリアに到達したとき及び停止したときにスレッド ID 独立な変数の全スレッドで値が一致すること) を証明する．最後にそれら 2 つの補題を用いて並列実行のための推論規則の健全性を証明する．

定理 9 の証明は補題 10 と同様にできるので，本稿では証明を省略し，以下では補題 10 を示す．本節ではバリア仕様 BS は precise であり，$\forall b, \bigstar_{i\in\mathsf{Tid}} BS(i,b)_{pre} \Rightarrow \bigstar_{i\in\mathsf{Tid}} BS(i,b)_{post}$ を満たすとする．

2.3.1　逐次実行についての健全性

逐次実行についての健全性は以下の補題である．

補題 11 (逐次実行についての健全性)**.** $BS, i \vdash_{seq} \{P\}\ C\ \{Q\} \Rightarrow BS, i \models_{seq} \{P\}\ C\ \{Q\}$

証明は $BS, i \vdash_{seq} \{P\}\ C\ \{Q\}$ の導出についての帰納法を行い，その上で各規則について $\models_{seq}$ の定義を展開し，Tsafe 述語の添字 n についての帰納法を行う．Barrier 規則以外は Vafeiadis と同様に証明できる．ここでは Barrier 規則のみを示す．以下の補題を示しておく．

補題 12. 任意の n, BS, i, s, h について，$s, h \models P$ ならば $\mathsf{Tsafe}_{i,n}((\mathbf{skip}, s), h, P, BS)$

証明. n について $n = 0$ のときは明らか．$n+1$ のときを考える．Tsafe の各条件を示す． (1) は $s, h, \models P$ より成り立つ．その他も簡単に示せる． □

補題 13 (Barrier 規則についての健全性)**.** 任意の BS, i, b, s, h, n について $s, h \models BS(i,b)_{pre}$ ならば $\mathsf{Tsafe}_{i,n}((\mathbf{barrier}_b, s), h, BS(i,b)_{post}, BS)$

証明. n についての帰納法による．$n = 0$ のときは明らか．$n = k+1$ の時を考える．Tsafe の各条件を示す．(1) , (2) , (3) , (4) は明らか．(5) を示す．$\mathsf{wait}(\mathbf{barrier}_b) = (b, \mathbf{skip})$．$h_p$ を h とし，h_F を $\forall l, h_F(l) = \bot$ となる pheap とする．$h_p \bot h_F$，$h = h_p \uplus h_F$ かつ $s, h_p \models BS(i,b)_{pre}$ は明らか．h_Q を $h_Q \bot h_F$ かつ $s, h_Q \models BS(i,b)_{post}$ を満たす任意のヒープとする．$h_Q \uplus h_F = h_Q$ より，$\mathsf{Tsafe}_{i,k}(\mathbf{skip}, s, h_Q, BS(i,b)_{post}, BS)$ を示せばよいが，$s, h_Q \models BS(i,b)_{post}$ と補題 7 より成り立つ． □

2.3.2　スレッド ID 独立性の健全性

長さ n_t の列 $\overline{C}$，$\overline{s}$，$\overline{h}$ について，カーネルの初期状態を表す述語 $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ を定義する．

定義 14. $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ は以下が成り立つとき，かつそのときに限って成り立つ．

- $\exists C_{init}\ \tau, (\forall i \in \mathsf{Tid}, C_i = C_{init}) \land \Gamma \vdash C_{init} : \tau$
- $\forall i, j \in \mathsf{Tid}, \Gamma \models s_i =_L s_j$
- $\biguplus_{i \in \mathsf{Tid}} h_i$ が定義できる
- $\forall i \in \mathsf{Tid}, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$

スレッド ID 独立性の健全性は以下の補題である．

補題 15. $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ かつ $(\overline{C}, \overline{s}, \biguplus_{i \in \mathsf{Tid}} h_i) \to_g^* (\overline{C'}, \overline{s'}, h')$ ならば以下が成り立つ．

1. $(\forall i \in \mathsf{Tid}, C'_i = \mathbf{skip})$ ならば $(\forall i, j \in \mathsf{Tid}, \Gamma \models s'_i =_L s'_j)$
2. $\forall i \in \mathsf{Tid}, \mathsf{wait}(C'_i) = (b_i, C''_i)$ ならば $(\forall i, j \in \mathsf{Tid}, b_i = b_j \land s'_i =_L s'_j)$

(1) は停止したときに型 Lo を持つ変数の値が全スレッドで一致していること，(2) はバリアに到達したときに型 Lo を持つ変数の値と到達したバリアの添字 (b_i) が全スレッドで一致していることを表す．

補題 15 の証明のために，まず以下の補題を示す．

補題 16 (non-interference)**.** $\Gamma \vdash C : \tau$，$s_1 =_L s_2$，$h_1 \perp h_2$，$(C, s_1, h_1) \to_p^* (C_1, s'_1, h'_1)$ かつ $(C, s_2, h_2) \to_p^* (C_2, s'_2, h'_2)$ ならば以下が成り立つ．

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

ここで $\to_p$ は $\to_t$ を heap から pheap へ拡張した意味論であり，以下のように定義される．

定義 17. $(C, s, h) \to_p (C', s', h')$ は以下が成り立つとき，かつそのときに限って成立する．

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. ある $h \perp h_F$ かつ $\mathsf{hdef}(h \uplus h_F)$ を満たす pheap h_F が存在して，$(C, s, h \uplus h_F) \to_t (C', s', h' \uplus h_F)$

reads はコマンドが次の 1 ステップの実行で読み込み命令を実行するならばその読み込みを行うアドレスを返す補助関数である．

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

補題 16 の証明は省略する．

補題 16 をカーネル全体の意味論に適用するために，以下の補題を示す．

補題 18. $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, BS)$ かつ $(\overline{C}, \overline{s}, \biguplus_{i \in \mathsf{Tid}} h_i) \to_g^* (\overline{C'}, \overline{s'}, h')$ ならばある $\overline{C''}, \overline{s''}, \overline{h''}$ が存在して，$\mathsf{initConf}(\overline{C''}, \overline{s''}, \overline{h''}, \Gamma, BS)$ かつ $\forall i \in \mathsf{Tid}, (C''_i, s''_i, h''_i) \to_p^* (C'_i, s'_i, h'_i)$

証明. 証明 $\to_g^*$ に関する帰納法による． □

補題 16 と補題 18 を用いると，補題 15 は以下のように示せる．

証明. 仮定に補題 18 を適用すると， ある $\overline{C''}, \overline{s''}, \overline{h''}$ が存在して，$\mathsf{initConf}(\overline{C''}, \overline{s''}, \overline{h''}, \Gamma, BS)$ かつ $\forall i \in \mathsf{Tid}, (C''_i, s''_i, h''_i) \to_p^* (C'_i, s'_i, h'_i)$ が成り立つ．ここで C_{init} を $\forall i \in \mathsf{Tid}, C''_i = C'_{init}$ をみたすコマンドとする．任意の i, j に対して補題 16 を $(C_{init}, s''_i, h''_i) \to_p^* (C'_i, s'_i, h_i)$ と $(C_{init}, s''_j, h''_j) \to_p^* (C'_j, s'_j, h'_j)$ に適用すると結論が得られる． □

2.3.3　並列実行に関する CSL の健全性

補題 11 と補題 15 を用いて定理 9 を証明する．まず以下の補題を示す．

補題 19. 以下が成り立つとする．

(a) $h = \uplus_{i\in\mathsf{Tid}} h_i$
(b) $\forall i \in \mathsf{Tid}, \mathsf{Tsafe}_{i,n}(C, s_i, h_i, Q_i, BS)$
(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) $\bigstar_{i\in\mathsf{Tid}} Q_i \Rightarrow Q$

このとき $\mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$

証明. n に関する帰納法で証明する．$n = 0$ のときは明らか．$n = n' + 1$ のときを考える．Gsafe_k の各条件を証明する．

1. $\forall i \in \mathsf{Tid}, C_i = \textbf{skip}$ とする．このとき (b) より $\forall i \in \mathsf{Tid}, s_i, h_i \models Q_i$．(c) 及び (d) に補題 15 を適用すると $\forall i, j \in \mathsf{Tid}, s_i =_L s_j$．よって $\overline{s}, h \models \bigstar_{i\in\mathsf{Tid}} Q_i$．(e) より $\overline{s}, h \models Q$
2. 任意の i' について，$h' = (\uplus_{i\in\mathsf{Tid}\wedge i\neq i'} h_i) \uplus h_F$ とする．(b) の $i = i'$ の場合の (2) を $h_F = h'$ として適用すると，$h_i \uplus h' = h \uplus h_F$ より $(C_i, s_i, h \uplus h_F) \nrightarrow_t \textbf{abort}$．$i'$ は任意だったので $(\overline{C}, \overline{s}, h \uplus h_F) \nrightarrow_g \textbf{abort}$.
3. (1) の場合と同様に補題 15 を適用すればよい．
4. $(\overline{C}, \overline{s}, h \uplus h_F) \rightarrow_g (\overline{C'}, \overline{s'}, h')$ として，これを導く規則について場合分けを行う．(G-Step) のとき 1 ステップ進めるスレッドの ID を k とし，$(C_k, s_k, h \uplus h_F) \rightarrow_t (C'_k, s'_k, h')$ とする．このとき k 以外の i については $C'_i = C_i, s'_i = s_i$．$h'_F = (\uplus_{i\in\mathsf{Tid}, i\neq k} h_i) \uplus h_F$ とすると $h \uplus h_F = h_k \uplus h'_F$．ここで (b) の $i := k$ の場合の (4) をもちいると，ある h''_k が存在して，$h' = h''_k \uplus h'_F$ かつ $\mathsf{Tsafe}_{k,n'}(C'_k, s'_k, h''_k, Q_k)$．また，$k$ 以外の i に対して，$h''_i = h_i$ とすると (b) より $\mathsf{Tsafe}_{i,n'}(C'_i, s'_i, h''_i, Q_i)$．よって $\forall i \in \mathsf{Tid}, \mathsf{Tsafe}_{i,n'}(C'_i, s'_i, h''_i, Q_i)$．ここで $h'' = \uplus_{i\in\mathsf{Tid}} h''_i$ とすると $h' = h'' \uplus h_F$ であり，帰納法の仮定を適用すると $\mathsf{Gsafe}_{n'}(\overline{C'}, \overline{s'}, h'', Q, \Gamma)$ が示せる．(G-Barrier) の場合は省略するが，同様に示せる．

□

補題 19 を用いて定理 9 を証明する．

証明. 仮定 $\Gamma, BS \vdash_{par} \{P\}\ C\ \{Q\}$ より以下が成り立つ．

1. $P \Rightarrow \bigstar_{i\in\mathsf{Tid}} P_i$
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. $\forall b,\ \bigstar_{i\in\mathsf{Tid}} BS(i,b)_{pre} \Rightarrow \bigstar_{i\in\mathsf{Tid}} BS(i,b)_{post}$

このとき，任意の $\overline{s}$ と h について $\forall i \in \mathsf{Tid}, s_i(\mathtt{tid}) = i$，$\forall i, j \in \mathsf{Tid}, s_i =_L s_j$ かつ $\overline{s}, h \models P$ ならば $\forall n, \mathsf{Gsafe}_n(\overline{C}, \overline{s}, h, Q, \Gamma)$ を示す．(1) より $\exists \overline{h}, h = \uplus_i h_i$ かつ $\forall i, s_i, h_i \models P_i$．これと (3) より $\forall i, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$．また明らかに $(\overline{C}, \overline{s}, h) \rightarrow_g^* (\overline{C}, \overline{s}, h)$．ここで補題 19 を適用すれば結論が導ける．

□

2.4　Blom の CSL との違い

ここでは Blom らの研究と本研究をまず CSL の導出規則に関して，次に証明方針について比較する．

2.4.1　推論規則の違い

Blom らの CSL と GPUCSL の違いとして， (i) 言明の形式と (ii) Frame 規則の有無が挙げられる．

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

図 2.16: Blom らの CSL の Write 規則

(i) Blom らの推論規則は Coq 上での健全性証明に適さない．Blom らの CSL の言明はリソースに関する言明と仕様に関する言明からなる．Blom らの CSL の仕様は $\{R_{pre}, P_{pre}\}\ C\ \{R_{post}, P_{post}\}$ である．R_{pre}, R_{post} はリソースに関する言明であり，P_{pre}，P_{post} は仕様に関する言明である．

図 2.16 が Blom らの CSL の Write 規則である．リソースに関する言明に現れる $\mathsf{LPerm}(e, \mathsf{rw})$ は GPUCSL の $\exists e', e \mapsto^1 e'$ と等しい．仕様に関する言明に現れる $L[e_1]$ はアドレス e_1 の指す値を表す．Blom らの CSL の証明規則は，仕様に関する言明が言及する全てのメモリアドレスが，リソースに関する言明で言及されていることを前提として要求する．この条件は比較的複雑であるため，Coq での健全性証明を複雑にさせると考えられる．

(ii) Blom らの CSL には Frame 規則が無いが，GPUCSL は標準的な CSL に近い設計を目標としたので，Frame 規則を持つ．関数の仕様をモジュラーに記述するには Frame 規則が必要になるが，後述するように Frame 規則で拡張した CSL に Blom らの健全性の証明を単純に適用することはできない．

2.4.2　健全性証明の問題点

Blom らの健全性証明に (i) Frame 規則の追加が難しい，(ii) スレッド ID 独立性のために必要な前提が欠けていることを明らかにした．

(i) Blom らの健全性証明は以下の補題 20 に依存している．[1]

補題 20. $\Gamma \vdash_B \{P\}\ C\ \{Q\}$，$\overline{s}, h \models P$，$(\overline{C}, \overline{s}, h) \to_g^* (\overline{C'}, \overline{s'}, h')$ かつ $\forall i \in \mathsf{Tid}, \mathsf{wait}(C_i') = (b, C_i'')$ ならば $\overline{s'}, h' \models \bigstar_{i \in \mathsf{Tid}} BS(i, b)_{pre}$

この補題は，$\{P\}\ C\ \{Q\}$ であるカーネル C がバリアに到達したとき，到達したときのメモリ状態がバリア仕様の事前条件を満たすことを意味する．しかしこの補題は Frame 規則を持つ CSL では一般に成り立たない．以下のカーネル C を考える．

```
1 Gl arr[tid] = tid;
2 barrier0;
3 Gl arr[tid] = tid;
```

GPUCSL では，以下の言明のもとで $\Gamma \vdash_B \{P\}\ C\ \{Q\}$ が導出できる．

- $P := \bigstar_{i \in \mathsf{Tid}}(\exists v, \mathbf{Gl}\ \mathtt{arr}[i] \mapsto v)$
- $Q := \bigstar_{i \in \mathsf{Tid}}(\mathbf{Gl}\ \mathtt{arr}[i] \mapsto i)$
- $P_i := \exists v, \mathbf{Gl}\ \mathtt{arr}[i] \mapsto v$
- $Q_i := \mathbf{Gl}\ \mathtt{arr}[i] \mapsto i$
- $\forall i, BS(i, 0) := \mathbf{emp}$

任意の i について，$\mathsf{BS}, i \vdash_T \{P_i \wedge \mathtt{tid} = i\}\ C\ \{Q_i\}$ は以下のように示せる．

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

[1] 補題 20 は文献 [10] の以下の文に対応する．“Since the barrier resources properly divide the group resources, the resources required by the second part of the trace are available.”

10 $\{\mathbf{Gl}\ \mathtt{arr}[i] \mapsto \mathtt{tid} \wedge \mathtt{tid} = i\} \Rightarrow$
11 $\{\mathtt{a} + i \mapsto i\}$

6 行目の導出で Frame 規則を用いていることに注意せよ．ここでバリアに到達した時のカーネルのヒープ h は $\forall i \in \mathsf{Tid}, h(i) = i$ なので明らかに $\overline{s}, h \models \bigstar_{i \in \mathsf{Tid}} \mathbf{emp}$ $(= \mathbf{emp})$ を満たさない．以上の議論から Blom らの CSL の証明は Frame 規則を持つ GPUCSL の証明には適さないことが分かる．

(ii) 我々は補題 15 でスレッド ID 独立なカーネルがバリア相違を起こさないことを証明した．補題 15 では Blom らが言及していない条件 $\mathsf{initConf}(\overline{C}, \overline{s}, \overline{h}, \Gamma, \mathsf{BS})$，特に $\forall i \in \mathsf{Tid}, n, \mathsf{Tsafe}_{i,n}(C_i, s_i, h_i, Q_i, BS)$ を仮定しているが，これはカーネルが競合状態を起こさないことを意味し，この条件は外すことができない．なぜなら以下のようにスレッド ID 独立ではあるが，競合状態が発生するカーネルで，バリア相違を起こす反例が存在するからである．

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

このカーネル C は $\Gamma(\mathtt{a}) = \mathsf{Lo} \wedge \Gamma(\mathtt{x}) = \mathsf{Lo}$ となる Γ のもとで $\Gamma \vdash C : \mathsf{Lo}$ が導出できるが，3 行目の条件式の評価結果はスケジューリングによって異なるので，このカーネルはバリア相違を起こすことがある．

2.5　応用例

GPUCSL の検証例として図 2.2 の `stencil` カーネルの検証を示す．`stencil` カーネルの仕様を以下のように与える．[2]．

$$\begin{aligned}&\forall f.\ \vdash_G \{\mathsf{array}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star \mathsf{array}(\mathbf{Gl}\ \mathtt{out}, n_t n_b, -)\}\\ &\qquad \mathtt{stencil}\\ &\qquad \{\mathsf{array}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star \mathsf{array}(\mathbf{Gl}\ \mathtt{out}, n_t n_b, \lambda k. f'(k-1) + f'(k) + f'(k+1))\}\end{aligned}$$

ここで $f'(k) \coloneqq \mathbf{if}\ k < 0 \vee n_t n_b \le k\ \mathbf{then}\ 0\ \mathbf{else}\ f(k)$ とする．スレッド i, j の事前条件，事後条件及び 0 番目のバリア同期命令のバリア仕様をそれぞれ $P_{i,j}$，$Q_{i,j}$，$BS(i,j)$ として，以下のように定義する．

$$\begin{aligned}P_{i,j} \coloneqq\ & \boxed{\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star (\mathbf{Gl}\ \mathtt{out}[i + jn_t] \mapsto^1 -)} \star\\ & (\mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^1 -) \star\\ & (\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{Sh}\ \mathtt{smem}[0] \mapsto^1 -\ \mathbf{else}\ \mathbf{emp}) \star\\ & (\mathbf{if}\ i = n_t - 1\ \mathbf{then}\ \mathbf{Sh}\ \mathtt{smem}[n_t+1] \mapsto^1 -\ \mathbf{else}\ \mathbf{emp})\\ Q_{i,j} \coloneqq\ & \boxed{\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star \mathbf{Gl}\ \mathtt{out}[i + jn_t] \mapsto^1 f'(i + jn_t - 1) + f'(i + jn_t) + f'(i + jn_t + 1) \star}\\ & \mathsf{array}_{1/(n_t n_b)}(\mathbf{Sh}\ \mathtt{smem}, n_t + 2, \lambda k. f'(k + jn_t - 1))\\ \mathsf{BS}(i,j)_{pre} \coloneqq\ & (\mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^1 f'(i + jn_t)) \star\\ & (\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{Sh}\ \mathtt{smem}[0] \mapsto^1 f'(jn_t - 1)\ \mathbf{else}\ \mathbf{emp}) \star\\ & (\mathbf{if}\ i = n_t - 1\ \mathbf{then}\ \mathbf{Sh}\ \mathtt{smem}[n_t+1] \mapsto^1 f'((j+1)n_t)\ \mathbf{else}\ \mathbf{emp})\\ \mathsf{BS}(i,j)_{post} \coloneqq\ & \mathsf{array}_{1/(n_t n_b)}(\mathbf{Sh}\ \mathtt{smem}, n_t + 2, \lambda k. f'(k + jn_t - 1))\end{aligned}$$

$P'_{i,j}$，$Q'_{i,j}$ をそれぞれ $P_{i,j}$，$Q_{i,j}$ の $\boxed{\cdots}$ で囲われた式とする．つまり $P'_{i,j}$，$Q'_{i,j}$ はそれぞれ $P_{i,j}$，$Q_{i,j}$ のうち，共有メモリに言及しない言明である．ブロック j の事前条件 P_j，事後条件 Q_j を $\bigstar_{i \in \mathsf{Tid}} P'_{i,j}$，$\bigstar_{i \in \mathsf{Tid}} Q'_{i,j}$ とする．stencil が仕様を満たすことを証明するには，以下を示せば良い．

[2] 事後条件 $\{\mathsf{array}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star \ldots\}$ に出現するプログラム変数 `arr` などは実際の証明では $\forall f, l_{in}, l_{out}.\ \vdash_G \{\mathtt{arr} = l_{in} \wedge \mathtt{out} = l_{out} \wedge \ldots\}\{\ldots\}\{\ldots\}$ のように束縛された変数を用いて書かれた条件 $\{\mathsf{array}(\mathbf{Gl}\ l_{arr}, n_t n_b, f) \star \ldots\}$ の略記である．

- $P \Rightarrow \bigstar_{j \in \mathsf{Bid}} P_j$ (Grid 規則の 1 行目)
- $\bigstar_{j \in \mathsf{Bid}} Q_j \Rightarrow Q$ (Grid 規則の 3 行目)
- $\forall j. P_j \star \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, n_b + 2, -) \Rightarrow \bigstar_{i \in \mathsf{Tid}} P_{i,j}$ (Block 規則の 1 行目)
- $\forall ij. \mathsf{BS}, i \vdash_T \{\mathtt{tid} = i \wedge \mathtt{bid} = j \wedge P_{i,j}\}\mathtt{stencil}\{Q_{i,j}\}$ (Block 規則の 2 行目)
- $\forall j. \bigstar_{i \in \mathsf{Tid}} Q_{i,j} \Rightarrow Q_j \star \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, n_b + 2, -)$ (Block 規則の 3 行目)
- $\forall j. \bigstar_{i \in \mathsf{Tid}} \mathsf{BS}_{pre}(i, j) \Rightarrow \bigstar_{i \in \mathsf{Tid}} \mathsf{BS}_{post}(i, j)$ (Block 規則の 4 行目)

これらの条件は容易に示すことができる.

第3章　GPGPUカーネル検証のためのライブラリ: GPUVeLib

本節ではGPGPUプログラム検証を行うためのCoqライブラリGPUVeLibを提案する．

2節で与えたGPUCSLの健全性証明の際に作成したCoq上での形式化を用いればGPGPUカーネルの正しさをCoq上で検証できる．しかし，GPUCSLの規則を直接適用するスタイルの証明には証明スクリプトが冗長になる問題がある．例えば，以下の3つ組を考える (事後条件は重要ではないので省略している).

1 $\{(\mathtt{ix} = x * n + \mathtt{tid}) \wedge \mathsf{array}_{1/n}(\mathtt{in}, 0, \mathtt{ix}, f) \star \mathbf{Gl}\ \mathtt{in}[\mathtt{ix}] \mapsto f(i) \star \ldots\}$
2 $\quad \mathtt{t} := \mathbf{Gl}\ \mathtt{in}[\mathtt{ix}]$
3 $\{\ldots\}$

この3つ組を証明するには (i) 事前条件の $\mathbf{Gl}\ \mathtt{in}[\mathtt{ix}] \mapsto f(i)$ が分離積の最も左に出現するように変形し，(ii) Frame規則を適用してそれ以外の言明を無視してから，(iii) Read規則を適用する必要がある．このような事前条件の書き換えや明示的なFrame規則の適用は証明スクリプトを煩雑にしてしまう．

本研究ではそのような冗長な証明スクリプトを避けるために，証明を支援するタクティックライブラリGPUVeLibを設計，実装した．GPUVeLibはCoqのタクティック作成のための言語Ltacを用いて実装した．タクティックの設計はCaoら [12] やChlipala [14] による逐次プログラム検証のための分離論理による検証ライブラリを参考にした．これらの検証ライブラリはほぼ全ての証明義務を自動的に証明することを目標に設計されているが，本研究では簡単な証明義務のみを自動的に証明するように設計した．そのような設計を行った理由として，これらの全ての証明義務を解くライブラリの実装は容易ではなく，また本研究で対象とするコードテンプレートは比較的小規模であるため，自動化がそれほど重要ではない点が挙げられる．また，GPUVeLibは自動検証タクティックによって証明できない部分の証明を補助するためのタクティックももつ．

本研究では，GPUVeLibの有用性を確認するためにいくつかのGPGPUカーネルコードに対して検証を行った．検証を行ったGPGPUカーネルはGPUDSLで用いられるコードテンプレートをメタ言語コードを含まないように単純化したものである．検証を行ったカーネルは以下の3つである．

- 入力配列の全要素に1を足した新しい配列を求めるカーネル (mapカーネル)
- 入力配列の全て要素の和を並列に求めるカーネル (reduceカーネル)
- 入力配列の接頭辞和列を並列に求めるカーネル (prescanカーネル)

Reduceカーネルやprescanカーネルはバリア同期を用いて複雑なリソース交換を行うが，GPUVeLibを用いることでそれらのカーネルに対しても検証を行うことができた．

本章は以下のような構成から成る．まず3.1節でタクティックによる証明の概略を示す．3.2.1節ではGPUVeLibの設計を示す．3.4節ではいくつかのGPGPUカーネルについてGPUVeLibを用いて検証した結果を示す．

3.1　Coqと Ltac言語

本節ではCoqによる証明の概略を示す．

Coqは対話的な証明環境であり，ユーザは現在示すべき目標と現在使うことが可能な仮定を見ながら証明を進めていく．証明はタクティックという証明記述のための命令群を用いて行う．タクティックの中には現在の仮定やすでに証明された補題を仮定，ゴールに適用する `apply` タクティックや，証明された

```
1   n : nat                          1   n : nat
2   m : nat                          2   m : nat
3   H : m = 0                        3   H : m = 0
4  ========================         4  ========================
5   n = n + m                        5   n = n + 0
```

初期状態　　`rewrite H` の実行後

図 3.1: Coq による証明例

タクティック	目的
hoare_forward	$\mathsf{BS}, i \vdash_T \{P\}\, C_1; C_2\, \{Q\}$ に対して，C_1 で必要になる言明を P から探し，適切な導出規則を適用して $\mathsf{BS}, i \vdash_T \{P'\}\, C_2\, \{Q\}$ に変形する
sep_cancel	$s, h \models P \Rightarrow Q$ において P と Q の両方に共通して現れる言明を削除する
sep_rewrite	$p \Leftrightarrow q$ を用いて $\mathsf{BS}, i \vdash_T \{P\}\, C\, \{Q\}$ 中に出現する p を q に置き換える

表 3.1: GPUVeLib の概要

等式 $x = y$ で仮定，ゴール中の x を y に置き換えるタクティック `rewrite` タクティックなどがある．図 3.1 に Coq による証明例を示す．ここでは簡単な定理 $\forall n, m. m = 0 \Rightarrow n = n + m$ の証明を示す．二重線の上が現在の仮定群，下が現在のゴールを示している．仮定は現在自然数の変数 `n`，`m` 及び仮定 $\mathtt{m} = 0$ が使えることを示している．図 3.1 左の初期状態に対して `rewrite H` を実行すると，現在のゴールに出現する `m` が 0 に置き換わる．最後に定理 `plus_n_0` : $\forall n, n + 0 = n$ に `apply` タクティックを適用し実行すると証明が完了する．

その他のタクティックとして，Coq は整数の等式，不等式を自動証明する `omega`，`ring`，`nia` タクティックをもつ．

3.2　GPUVeLib

GPUVeLib は Coq のタクティックライブラリであり，ユーザ定義タクティックを作成するための言語 Ltac を用いて実装した．表 3.1 はタクティックの概略である．GPUVeLib は主に自動証明のためのタクティック hoare_forward，sep_cancel，および手動証明を補助するタクティック sep_rewrite を持つ．hoare_forward は現在の事前条件を利用してプログラムを 1 ステップ記号実行するタクティックである．sep_cancel はゴール $s, h \models P \Rightarrow Q$ を自動証明するためのタクティックであり，P と Q の両方に出現する言明を削除することでゴールを単純化する．sep_rewrite は $p \Leftrightarrow q$ が成り立つときに 3 つ組中に出現する言明 p を q に置き換えるためのタクティックである．これは array や sarray に対して，具体的な言明 $L \mapsto E$ を取り出すときに用いる．

3.2.1　自動証明タクティック

自動検証のためのタクティック (sep_cancel，hoare_forward) について説明する．sep_cancel は含意 $s, h \models P \Rightarrow Q$ に対して，P と Q 両方に出現する言明を消去することで含意を単純化するタクティックである．hoare_forward はゴール $BS, i \vdash_{seq} \{P\}\, C\, \{Q\}$ に対して，P の下で C を 1 ステップ記号実行して新しいゴールに変形するタクティックである．GPUVeLib を利用するユーザは，ゴール $BS, i \vdash_{seq} \{P\}\, C\, \{Q\}$ に対して，hoare_forward を繰り返し適用し，最後に出てきた含意 $s, h \models P' \Rightarrow Q'$ に対して sep_cancel を繰り返し適用することで，証明を完了させることができる．hoare_forward と sep_cancel の簡単な動作例を図 3.2 と図 3.3 に示す．

図 3.2 の左に hoare_forward タクティックの適用前の状態を，右に適用後の状態を示した．hoare_forward の適用前の 3 つ組において，次に実行するコマンドが $T := [\mathbf{GI}\ X]$ である．hoare_forward は次に実行する

```
1  ntrd : nat
2  tid : Fin.t ntrd
3  tx : val
4  ty : val
5  ============================
6  BS, tid ⊢seq
7    {Gl Y ↦ ty ⋆ Gl X ↦ tx}
8    T := [Gl X];
9    U := [Gl Y];
10   [Gl X] := U;
11   [Gl Y] := T;
12   {Gl X ↦ tx ⋆ Gl Y ↦ ty}
```

hoare_forward 適用前

```
1  ntrd : nat
2  tid : Fin.t ntrd
3  tx : val
4  ty : val
5  ============================
6  BS, tid ⊢seq
7    {Gl X ↦ tx⋆!(T == tx) ⋆ Gl Y ↦ ty}
8    U := [Gl Y];
9    [Gl X] := U;
10   [Gl Y] := T;
11   {Gl X ↦ tx ⋆ Gl Y ↦ ty}
12
```

hoare_forward 適用後

図 3.2: hoare_forward の実行例

```
1  s : stack
2  h : pheap
3  HP : s ⊨ (U == ty)
4  HP0 : s ⊨ (T == tx)
5  H : s, h ⊨ Gl Y ↦ T ⋆ Gl X ↦ U
6  ============================
7  s, h ⊨ Gl  ↦ ty ⋆ Gl Y ↦ tx
```

sep_cancel 適用前

```
1  s : stack
2  h : pheap
3  HP : s ⊨ (U == ty)
4  HP0 : s ⊨ (T == tx)
5  H : s, h ⊨ Gl Y ↦ T
6  ============================
7  s, h ⊨ Gl Y ↦ tx
```

sep_cancel 適用後

図 3.3: sep_cancel の実行例

コマンドが必要とするリソースを調べ，それが事前条件に含まれるか検査する．例の場合は T := [**Gl** X] は **Gl** X にアクセスするため，**Gl** $X \mapsto ?$ という形の言明を事前条件から探す．次にそのリソースを表す言明が最も左に現れるように事前条件を変形した上で Frame 規則を適用することで次のコマンドの実行に関係がない言明を無視し，対応する規則を適用する．例の場合は **Gl** $X \mapsto tx$ を事前条件の左に移動し，Frame 規則を適用することで $Gl\ Y \mapsto ty$ を無視し，Read 規則を適用する．Frame 規則や Read 規則は前提に変数に関する副条件が存在するが，それらの条件も自動的に証明する．

図 3.3 の左に sep_cancel タクティックの適用前の状態を，右に適用後の状態を示した．sep_cancel タクティックは結論 $s, h \models Q_1 \star Q_2 \star \cdots \star Q_n$ に対して，$s, h \models P_1 \star P_2 \star \ldots P_m$ という形の仮定を文脈から探す．次に $s, h \models P_i \Rightarrow Q_1$ が導けるような $1 \leq i \leq m$ を探し，ゴールと仮定からそれぞれ Q_1，P_i を削除して単純化する．例の場合は Q_1 = **Gl** $X \mapsto ty$ であり，仮定 HP より $s, h \models$ **Gl** $X \mapsto U \Rightarrow$ **Gl** $\mapsto ty$ であるため，仮定 H と結論からそれぞれ **Gl** $X \mapsto U$ と **Gl** $X \mapsto ty$ を削除する．

3.3　手動証明を補助するタクティック

本節では sep_rewrite タクティックについて説明する．

sep_rewrite タクティックは関係 $P \Leftrightarrow Q$ を用いて，現在の文脈に出現する P を Q に置き換えるためのタクティックである．2 節で示した array_forward や sarray_forward などの補題に sep_rewrite を適用することで，事前条件の array や sarray から次のコマンドの実行に必要な言明を取り出すことができる．図 3.4 に sep_rewrite タクティックの実行例を示した．sep_rewrite の実行前は事前条件に array(**Gl** arr, l, f) があるが，**Gl** arr[i] $\mapsto$ – という形の言明がないため，hoare_forward タクティックは適用できない．そこで sep_rewrite タクティックを補題 array_forward とアクセスする添字 i に適用し実行することで，事前条件から次のコマンドの実行に必要な言明を取り出すことができる．

```
1 H : i < l
2 ============================
3 BS, tid ⊢_seq
4   {array(Gl arr, l, f) ⋆ ...}
5   X := Gl arr[i]; C
6   {...}
```

sep_rewrite 適用前

```
1 H : i < l
2 ============================
3 BS, tid ⊢_seq
4   {array(Gl arr, i, f) ⋆ Gl arr[i] ↦ f(i)⋆
5    array(Gl arr, i + 1, l − i − 1, f) ⋆ ...}
6   X := Gl arr[i]; C
7   {...}
```

sep_rewrite 適用後

図 3.4: sep_rewrite array_forward の実行例

3.4　事例研究

これらのカーネルの検証の概略を表 3.2 に示す．各カーネルに関して，左の列から順にカーネル名，

カーネル名	カーネルの行数	ループ不変式，バリア仕様の行数	証明の行数
map カーネル	5 行	9 行	230 行程度
reduce カーネル	10 行	55 行	900 行程度
prescan カーネル	23 行	88 行	1200 行程度

表 3.2: カーネル検証の概略

カーネルの行数，証明に必要になったループ不変式，バリア仕様の行数，証明スクリプトの行数を表す．

本研究では，GPUVeLib の有用性を確認するためにいくつかの GPGPU カーネルコードに対して検証を行った．検証を行った GPGPU カーネルは GPUDSL で用いられるコードテンプレートをメタ言語コードを含まないように単純化したものである．検証を行ったカーネルは以下の 3 つである．

- 入力配列の全要素に 1 を足した新しい配列を求めるカーネル (map カーネル)
- 入力配列の全て要素の和を求めるカーネル (reduce カーネル)
- 入力配列の接頭辞和列を求めるカーネル (prescan カーネル)

ここで列 $a_0, a_1, \ldots, a_i, \ldots, a_{n-1}$ の接頭辞和列とは $0, a_0, a_0 + a_1, \ldots, \sum_{j=0}^{i-1} a_j, \ldots, \sum_{j=0}^{n-2} a_j$ のことである．以下では各カーネルの検証の概略を示す．

3.4.1　事例研究 1: map カーネル

map カーネルは以下のカーネルである．

```
1 map() {
2   ix := gid;
3   while (i < len) {
4     tmp := Gl in[ix];
5     Gl out[ix] := tmp + 1;
6     ix := ix + n_t n_b;
7   }
8 }
```

in，out，len はそれぞれ入力配列，出力配列，配列の長さを表す変数である．map カーネルは入力配列と出力配列の長さが等しいことを要求する．map カーネル中の各スレッドは配列 in の $n_t n_b$ で割った剰余が gid と一致する位置の要素に対して，その値に 1 を足して配列 out の i 番目に書き込む．map カーネルの仕様を以下のように与える．

$$\forall l\ f.\ \vdash_G \{\texttt{len} = l \wedge \mathsf{array}(\mathbf{Gl}\ \texttt{in}, l, f) \star \mathsf{array}(\mathbf{Gl}\ \texttt{out}, l, -)\}$$

map

$$\{\mathsf{array}(\mathbf{Gl}\ \mathtt{in}, l, f) \star \mathsf{array}(\mathbf{Gl}\ \mathtt{out}, l, \lambda x.f(x)+1)\}$$

map カーネルの各スレッド i, j の事前条件，事後条件 $P_{i,j}$，$Q_{i,j}$ を以下のように定める．

$$P_{i,j} := \mathtt{len} = l \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, l, f) \star \mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, -, n_t n_b, i + n_t j)$$
$$Q_{i,j} := \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, l, f) \star \mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, \lambda x.f(x)+1, n_t n_b, i + n_t j)$$

各スレッドは，$n_t n_b$ 毎に離れた配列要素にアクセスを行うため，リソースの条件には述語 $\mathsf{sarray}(L, s, l, f, d, i)$ を用いる．

スレッドブロック，スレッドグリッド実行の検証における，事前条件，事後条件の集約については array_distribute，sarray_distribute を用いることで示すことができる．ここでは $\mathsf{BS}, i \vdash_T \{\mathtt{tid} = i \wedge \mathtt{bid} = j \wedge P_{i,j}\}\ \mathsf{map}\ \{Q_{i,j}\}$ を示す．これは以下のループ不変式のもとで証明できる．

$$\exists x.\mathtt{len} = l \wedge \mathtt{ix} = x n_t n_b + i + n_t j \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, l, f) \star$$
$$\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, \lambda k.\mathbf{if}\ k < x n_t n_b + i + n_t j\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)$$

上のループ不変式を $\exists x.I(x)$，map カーネルのループの中を C_{body} とおくと，以下を示せばよい．

1. $P_{i,j} \wedge \mathtt{ix} = i + j n_t \Rightarrow I(0)$
2. $\{I(x) \wedge (\mathtt{ix} < \mathtt{len})\}\ C_{\mathsf{body}}\ \{I(x+1)\}$
3. $(\exists x.I(x)) \wedge (\mathtt{ix} \geq \mathtt{len}) \Rightarrow Q_i$

1，3 は sarray_eq を用いることで示せる．2 は 4 行目，5 行目の配列読み込み，書き込みに対してそれぞれ array_forward，sarray_forward を用いて Read，Write 規則の事前条件の言明を取り出し，C_{body} の実行後の状態が $I(x+1)$ を導くことを示すために sarray_eq を用いればよい．

以下に 2 の証明の概略を示す．$ix(x) := x n_t n_b + i + n_t j$ であり，1 つ前の言明から変化した部分を青色で示している．

1 $\{\mathtt{ix} = ix(x) \wedge ix(x) < l \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, l, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\} \Rightarrow$
2 $\{\mathtt{ix} = ix(x) \wedge ix(x) < l \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x), f) \star \mathbf{Gl}\ \mathtt{in}[ix(x)] \mapsto^{1/(n_t n_b)} f(ix(x)) \star$
  $\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x)+1, l - ix(x) - 1, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\}$
3 `tmp := Gl in[ix];`
4 $\{\mathtt{ix} = ix(x) \wedge ix(x) < l \wedge \mathtt{tmp} = f(ix(x)) \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x), f) \star \mathbf{Gl}\ \mathtt{in}[ix(x)] \mapsto^{1/(n_t n_b)} f(ix(x)) \star$
  $\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x)+1, l - ix(x) - 1, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, l, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\} \Rightarrow$
5 $\{\mathtt{ix} = ix(x) \wedge ix(x) < l \wedge \mathtt{tmp} = f(ix(x)) \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x), f) \star \mathbf{Gl}\ \mathtt{in}[ix(x)] \mapsto^{1/(n_t n_b)} f(ix(x)) \star$
  $\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x)+1, l - ix(x) - 1, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, ix(x), \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j) \star$
  $\mathbf{Gl}\ \mathtt{out}[ix(x)] \mapsto - \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, ix(x)+1, l - ix(x) - 1, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\}$
6 `Gl out[ix] := tmp + 1;`
7 $\{\mathtt{ix} = ix(x) \wedge \mathtt{tmp} = f(ix(x)) \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x), f) \star \mathbf{Gl}\ \mathtt{in}[ix(x)] \mapsto^{1/(n_t n_b)} f(ix(x)) \star$
  $\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x)+1, l - ix(x) - 1, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, ix(x), \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j) \star$
  $\mathbf{Gl}\ \mathtt{out}[ix(x)] \mapsto f(ix(x)) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, ix(x)+1, l - ix(x) - 1, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\}$
8 `ix := ix + ` $n_t n_b$`;`
9 $\{\mathtt{ix} = ix(x+1) \wedge \mathtt{tmp} = f(ix(x)) \wedge \mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x), f) \star \mathbf{Gl}\ \mathtt{in}[ix(x)] \mapsto^{1/(n_t n_b)} f(ix(x)) \star$
  $\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{in}, ix(x)+1, l - ix(x) - 1, f) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, 0, ix(x), \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j) \star$
  $\mathbf{Gl}\ \mathtt{out}[ix(x)] \mapsto f(ix(x)) \star$
  $\mathsf{sarray}(\mathbf{Gl}\ \mathtt{out}, ix(x)+1, l - ix(x) - 1, \lambda k.\mathbf{if}\ k < ix(x)\ \mathbf{then}\ f(k)+1\ \mathbf{else}\ -, n_t n_b, i + n_t j)\} \Rightarrow$
10 $\{I(x+1)\}$

1 行目から 2 行目の導出は sep_rewrite を array_forward に適用することで，4 行目から 5 行目の導出は sarray_forward に適用することでそれぞれ示せる．9 行目から 10 行目の導出は結論に対して sep_rewrite タクティックを用いて array_forward，sarray_forward で書き換えを行い，sep_cancel タクティックを繰り返し用いることで証明できる．それ以外は hoare_forward タクティックを用いることで示せる．

3.4.2　事例研究 2: reduce カーネルの検証

Reduce カーネルは並列畳み込みアルゴリズムの CUDA 向け実装 [19] を参考に実装した．Reduce カーネルは以下のカーネルである．

```
1  smem[n_t]; // shared memory の確保
2  t1 := Gl in[gid];
3  ix := gid + n_t n_b;
4  while (ix < len) {
5    t2 := in[ix]; t1 := t1 + t2;
6    ix := ix + n_t n_b;
7  }
8  Sh smem[tid] := t1;
9  barrier_0;
10 st := n_t / 2;
11 while (0 < st) {
12   if (tid < st) {
13     t1 := Sh smem[tid]; t2 := Sh smem[tid + st] Sh smem[tid] := t1 + t2;
14   }
15   st := st / 2;
16   barrier_1;
17 }
18 if (tid == 0) {
19   t := Sh smem[0]; Gl out[bid] := t;
20 }
```

Reduce カーネルは長さ l の配列 in の要素の総和 s_{in} と同じ総和をもつ長さ n_b の配列 out を計算するカーネルである．事前条件として，$l \geq n_t n_b$ かつ n_t が 2 の冪であることを要求する．Reduce カーネルはまず 2 行目から 8 行目でスレッド i, j が $k \bmod n_t n_b = i + jn_t$ になる位置 k についての和を求め，共有メモリの自分のスレッド ID の位置に結果を書き込む．その後バリア同期をとり，10 行目から 17 行目でブロック内で共有メモリ上の値の和を求める．最後にブロック内の和を配列 out の自分のブロック ID の位置に結果を書き込む．

Reduce カーネルの仕様は以下のようになる．

$$\forall l, f, e.\ n_t n_b \leq l \wedge n_t = 2^e \Rightarrow \vdash_G \{\mathtt{len} = l \wedge \mathsf{array}(\mathtt{in}, l, f) \star \mathsf{array}(\mathtt{out}, n_b, -)\}$$
$$\mathtt{reduce}$$
$$\left\{\exists f'.\ \sum_{k=0}^{n_b-1} f'(k) = \sum_{k=0}^{l-1} f(k) \wedge \mathsf{array}(\mathtt{in}, l, f) \star \mathsf{array}(\mathtt{out}, n_b, f')\right\}$$

$e = 0$ のときは容易に証明できる．以下 $e > 0$ とする．$f'(i, j)$ を $k \bmod n_t n_b = i + jn_t$ となるような k に関する $f(k)$ の和，つまり

$$f'(i, j) := \sum_{k=0, k \bmod n_t n_b = i + jn_t}^{l-1} f(k)$$

とする．このとき $\sum_{j \in \mathsf{Bid}} \sum_{i \in \mathsf{Tid}} f'(i, j) = \sum_{i=0}^{l-1} f(i)$ が成り立つ．スレッド i, j の事前条件 $P_{i,j}$，事後条件 $Q_{i,j}$ を以下のように定める．

$$P_{i,j} := \mathtt{len} = l \wedge \mathsf{array}_{1/n_t n_b}(\mathbf{Gl}\ \mathtt{in}, l, f) \star (\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{Gl}\ \mathtt{out}[j] \mapsto -\ \mathbf{else}\ \mathbf{emp}) \star \mathbf{Sh}\ \mathtt{smem}[i] \mapsto -$$

$$Q_{i,j} := \mathsf{array}_{1/n_t n_b}(\mathbf{Gl}\ \mathtt{in}, l, f) \star \left(\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{Gl}\ \mathtt{out}[j] \mapsto \sum_{i \in \mathsf{Tid}} f'(i, j)\ \mathbf{else}\ \mathbf{emp}\right) \star \mathbf{Sh}\ \mathtt{smem}[i] \mapsto -$$

以下ではスレッド i, j の実行の検証とバリア仕様に関する条件の証明のみを示す．スレッドブロック，スレッドのリソースの集約に関する条件は容易に示せる．1 つ目のループのループ不変式を以下のように

定める (ループ中でアクセスしないリソースに関する言明は Frame 規則を用いて無視する).

$$I_0 := \exists x.\texttt{len} = l \land \texttt{ix} = xn_tn_b + i + jn_t \land \texttt{t1} = \sum_{k=0,k \bmod n_tn_b = i + jn_t}^{\min(ix(x),l)-1} f(k) \land \mathsf{array}_{1/n_tn_b}(\mathbf{Gl}\ \texttt{in}, l, f)$$

上のループ不変式を用いると，8 行目の直後で以下の言明が成り立つことが示せる．

$$\texttt{t1} = f'(i,j) \land \mathsf{array}_{1/n_tn_b}(\mathbf{Gl}\ \texttt{in}, l, f) \star (\mathbf{if}\ i = 0\ \mathbf{then}\ \mathbf{Gl}\ \texttt{arr}[j] \mapsto - \ \mathbf{else}\ \mathbf{emp}) \star \mathbf{Sh}\ \texttt{smem}[i] \mapsto f'(i,j)$$

$f''(s,i,j)$ を以下のように定める．

$$f''(s,i,j) := \sum_{k=0,k \bmod s=i}^{n_t-1} f'(k,j)$$

2 つ目のループのループ不変式を以下のように定める．

$$\begin{aligned} I_1 :=& \exists e_s.\texttt{len} = l \land \texttt{st} = \lfloor 2^{e_s-1} \rfloor \land 2^{e_s} \le n_t \land \\ & (\mathbf{if}\ i < \lceil 2^{e_s-1} \rceil\ \mathbf{then}\ \mathbf{Sh}\ \texttt{smem}[i] \mapsto f''(2^{e_s}, i, j) \star \\ & \qquad \mathbf{Sh}\ \texttt{smem}[i + \lceil 2^{e_s-1} \rceil] \mapsto f''(2^{e_s}, i + \lceil 2^{e_s-1} \rceil, j)\ \mathbf{else}\ \mathbf{emp}) \star \\ & \mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \texttt{smem}, 2 \cdot \lceil 2^{e_s-1} \rceil, n_t - 2 \cdot \lceil 2^{e_s-1} \rceil, -)\ \mathbf{else}\ \mathbf{emp} \end{aligned}$$

ここでも 1 つ目のループと同様にアクセスされないリソースは Frame 規則を用いて無視する．$\lfloor x \rfloor$，$\lceil x \rceil$ はそれぞれ x 以下の最大の整数、x 以上の最小の整数を表す．ここで $\mathsf{array}(\mathbf{Sh}\ \texttt{smem}, 2 \cdot \lceil 2^{e_s-1} \rceil, l - 2 \cdot \lceil 2^{e_s-1} \rceil, -)$ はループ本体中でアクセスされない smem の位置に関する言明であり，スレッド 0 に保持させている．また $2^{-1} = 1/2$ であり，$\lfloor 2^{-1} \rfloor = 0$，$\lceil 2^{-1} \rceil = 1$ であることに注意されたい．$\texttt{st} > 0$ ならば $e_s > 0$ であるため，$\lceil 2^{e_s-1} \rceil = \lfloor 2^{e_s-1} \rfloor = 2^{e_s-1}$ と一致する．

各バリア b のバリア仕様 $\mathsf{BS}(b,i)$ を以下のように定める．

$$\begin{aligned} \mathsf{BS}(0,i)_{pre} :=& \mathbf{Sh}\ \texttt{smem}[i] \mapsto f'(i,j) \\ \mathsf{BS}(0,i)_{post} :=& \mathbf{if}\ i < 2^{e-1}\ \mathbf{then}\ \mathbf{Sh}\ \texttt{smem}[i] \mapsto f'(i,j) \star \mathbf{Sh}\ \texttt{smem}[i + 2^{e-1}] \mapsto f'(i + 2^{e-1}, j)\ \mathbf{else}\ \mathbf{emp} \\ \mathsf{BS}(1,i)_{pre} :=& \exists e_s.\texttt{st} = \lfloor 2^{e_s-1} \rfloor \land \\ & (\mathbf{if}\ i < \lceil 2^{e_s-1} \rceil\ \mathbf{then}\ \mathbf{Sh}\ \texttt{smem}[i] \mapsto f''(\lceil 2^{e_s-1} \rceil, i, j) \star \\ & \qquad \mathbf{Sh}\ \texttt{smem}[i + \lceil 2^{e_s-1} \rceil] \mapsto f''(2^{e_s}, i + \lceil 2^{e_s-1} \rceil, j)\ \mathbf{else}\ \mathbf{emp}) \star \\ & \mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \texttt{smem}, 2 \cdot \lceil 2^{e_s-1} \rceil, n_t - 2 \cdot \lceil 2^{e_s-1} \rceil, -)\ \mathbf{else}\ \mathbf{emp} \\ \mathsf{BS}(1,i)_{post} :=& I_0 \end{aligned}$$

バリア仕様の条件，スレッド実行の検証はこれらの定義のもとで示せる．

3.4.3　事例研究 3: prescan カーネルの検証

Prescan カーネルは GPU Gems 39 章 [20] の実装を参考にモデル化した．また，検証は Chong らによる配列の各要素が 0 または 1 であるときの検証 [15] を参考に行った．Prescan カーネルは以下のカーネルである．

```
1 o := 1;
2 d := n_t;
3 while (0 < d) {
4   barrier_0;
5   if (Tid <C D) {
6     t1 :=  Sh smem[o * (2 * tid + 1) - 1];
7     t2 :=  Sh smem[o * (2 * tid + 2) - 1];
8     Sh smem[o * (2 * tid + 2) - 1] := t1 + t2;
```

```
 9    }
10    o := o * 2;
11    d := d / 2;
12  }
13  if (tid == 0) {Sh smem[2 * n_t - 1] := 0}
14  d := 1;
15  while (d < n_t * 2) {
16    o := o / 2;
17    barrier_1;
18    if (tid < d) {
19      t1 :=  Sh smem[o * (2 * tid + 1) - 1];
20      t2 :=  Sh smem[o * (2 * tid + 2) - 1];
21      Sh smem[o * (2 * tid + 1) - 1] := t2;
22      Sh smem[o * (2 * tid + 2) - 1] := t1 + t2;
23    }
24    d := d * 2
25  }
```

グローバルメモリからの入力配列の読み込みや結果の書き込みは本質的ではないので省略した．Prescan カーネルはブロック内で共有メモリ上の配列 smem の接頭辞和を求めるカーネルである．smem は長さ $2n_t$ の配列であり，n_t は 2 の冪であることが事前条件として要求される．

本研究ではブロック内の実行に関する検証のみを行った．グリッド実行に関しては容易にブロック毎の仕様を集約することで行うことができる．Prescan カーネルの仕様は以下のようになる．

$$\forall f, e, n_t = 2^e \Rightarrow \Gamma \vdash_B \{\mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, 2n_t, f)\}$$
$$\mathtt{prescan}$$
$$\left\{\mathsf{array}\left(\mathbf{Sh}\ \mathtt{smem}, 2n_t, \lambda k. \sum_{s=0}^{k-1} f(s)\right)\right\}$$

以下 $f_{ps}(k) \coloneqq \sum_{s=0}^{k-1} f(s)$ とする．

各スレッドの事前条件，事後条件 P_i, Q_i，1 つ目，2 つ目ののループ不変式 I_0, I_1，バリア仕様 BS を以下のように定める．

$$P_i \coloneqq \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, 2i, 2, f)$$
$$Q_i \coloneqq \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, 2i, 2, f)$$
$$f_{up}(o)(k) \coloneqq \sum_{s=k+1-m}^{k} f(s) \qquad (m = \min(\max(\{2^x \mid i+1 = 2^x t, t \in \mathbb{N}\}), o))$$
$$f_{down}(o)(k) \coloneqq \begin{cases} \sum_{s=0}^{k+1-o} f(s) & (k+1 \bmod o = 0) \\ \sum_{s=k+1-m}^{k} f(s) & (\text{otherwise}, m = \max(\{2^x \mid i+1 = 2^x t, t \in \mathbb{N}\})) \end{cases}$$
$$I_0 \coloneqq \exists e_o, e_d.((\mathsf{d} = 0 \wedge \mathsf{o} = 2n_t) \vee (\mathsf{d} = 2^{e_d} \wedge \mathsf{o} = 2^{e_o} \wedge e_d + e_o = e)) \wedge$$
$$(\mathbf{if}\ i < 2^{e_d+1}\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, i \cdot \max(2, 2^{e_o}), \max(2, 2^{e_o}), f_{up}(2^{e_o}))\ \mathbf{else\ emp})$$
$$I_1 \coloneqq \exists e_o, e_d.(d = 2^{e_d} \wedge o = 2^{e_o} \wedge 2^{e_d} \cdot 2^{e_o} = 2n_t) \wedge$$
$$(\mathbf{if}\ i < \lceil 2^{e_d-1} \rceil\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, i \cdot \min(d, 2) \cdot 2^{e_o}, \min(d, 2) \cdot 2^{e_o}, f_{down}(2^{e_o}))\ \mathbf{else\ emp})$$
$$\mathsf{BS}(0, i)_{pre} \coloneqq I_0$$
$$\mathsf{BS}(0, i)_{post} \coloneqq \exists e_o, e_d.((d = 0 \wedge o = 2n_t) \vee (d = 2^{e_d} \wedge o = 2^{e_o} \wedge e_d + e_o = e)) \wedge$$
$$(\mathbf{if}\ i < 2^{e_d+1}\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, i \cdot 2^{e_o+1}, 2^{e_o+1}, f_{up})\ \mathbf{else\ emp})$$
$$\mathsf{BS}(1, i)_{pre} \coloneqq \exists e_o, e_d.(d = 2^{e_d} \wedge o = 2^{e_o} \wedge 2^{e_d} \cdot 2^{e_o} = n_t) \wedge$$
$$(\mathbf{if}\ i < \lceil 2^{e_d-1} \rceil\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \mathtt{smem}, i \cdot \min(d, 2) \cdot 2^{e_o+1}, \min(d, 2) \cdot 2^{e_o+1}, f_{down}(2^{e_o+1}))\ \mathbf{else\ emp})$$
$$\mathsf{BS}(1, i)_{post} \coloneqq \exists e_o, e_d.(d = 2^{e_d} \wedge o = 2^{e_o} \wedge 2^{e_d} \cdot 2^{e_o} = n_t) \wedge$$

$$(\textbf{if } i < 2^{e_d} \textbf{ then } \mathsf{array}(\textbf{Sh } \mathtt{smem}, i \cdot 2^{e_o+1}, 2^{e_o+1}, f_{down}(2^{e_o+1})) \textbf{ else emp})$$

これらの言明のもとで prescan カーネルは検証できた.

第4章　メタ言語コードを含むコードテンプレートの検証

本節では，メタ言語コードを含むコードテンプレートの検証手法を提案する．GPUDSL コンパイラでは GPGPU コード生成を行うためにコードテンプレートを用いる．コードテンプレートとは入出力配列の型，スケルトンに渡す関数 (ユーザ関数) のコンパイル結果 (ユーザ関数コード)，GPU のパラメータなどでパラメタライズされた GPGPU コードである．GPUDSL コンパイラは提供するスケルトン毎にコードテンプレートを用意し，コードテンプレートに引数を適用することで GPGPU コードを生成する．

コードテンプレートには，それが用いられるデータの方やユーザがスケルトンに渡す関数によって後から埋められる穴が有るため，既存の完全なコードに対する検証技術を直接用いることができない．そこで本研究はそのようなコードテンプレートの「正しさ」を明確に定義する．さらにコードテンプレートを GPGPU コードを生成する Coq の関数として形式化し，その正しさを Coq の定理として証明する．

コードテンプレートの仕様とは，(i) ユーザ関数コードとコードテンプレートの変数が衝突せず，(ii) ユーザ関数コードが適当な事前条件のもとでユーザ関数 f を計算するならば，コードテンプレートに型とユーザ関数コードと適用した時の生成コードは，スケルトンにユーザ関数 f を適用した時の結果を計算を行うことである．(i) はプログラムの構文のみに依存する条件であるため，容易に Coq で表現することができる．(ii) と結論の「計算」に関する条件は GPUCSL の 3 つ組として表す．

コードテンプレートは入出力配列の型に応じて異なる GPGPU コードを生成する．タプルのような複合データ型の配列をサポートする GPUDSL コンパイラでは，タプルの配列をタプルの各要素ごとに独立した配列を用いて表現することが一般的である．そのため，配列アクセスを行うコードは次元と配列の添字から GPGPU コードを生成するようなメタ言語関数を用いて生成される．例えば，2 要素をもつタプルの配列 a の i 番目の要素に値 v_1 と v_2 からなるタプルを格納するコードは $a_0[i] := v_1; a_1[i] := v_2$ となる．ここで a_0，a_1 はそれぞれタプルの第 1 要素，第 2 要素が格納された配列である．コードテンプレートを検証するには，このような配列アクセスコードを生成するメタ言語関数に関しても同様に仕様を与え検証する必要がある．

本節ではまず本節で用いる記法の定義を導入し，次に配列アクセスコードを生成するメタ言語関数について議論する．次にユーザ関数コードの仕様について議論する．最後に事例研究として，Accelerate コンパイラの map テンプレートと reduce テンプレートの一部の検証例を示す．reduce テンプレートの検証では simulating function を用いて検証を行う．これはカーネルの計算を表現するメタ論理の関数であり，reduce テンプレートの計算の正しさの証明を simulating function の正しさに帰着させることができることを示す．

4.1　記法

本節では $\overline{X}$ でリスト $[X_0, X_1, \ldots, X_{n-1}]$ を表し，$\#\overline{X}$ でその長さ n を表す．集合 T と自然数 n に対して，T^n で各要素が T の長さ n のリスト全体の集合を表す．$\mathbf{Gl}\ \overline{E}$ 及び $\overline{L}[i]$ をそれぞれリスト $\mathbf{Gl}\ E_0, \mathbf{Gl}\ E_1, \ldots \mathbf{Gl}\ E_{n-1}$ 及び $L_0[i], L_1[i], \ldots L_{n-1}[i]$ とする ($\mathbf{Sh}\ \overline{E}$ についても同様)．また，長さの等しいリスト $\overline{L}$，$\overline{v}$ に対して，$\overline{L} \mapsto^p \overline{v}$ を $L_0 \mapsto^p v_0 \star L_1 \mapsto^p v_1 \star \cdots \star L_{n-1} \mapsto^p v_{n-1}$ と定義する．また，$\overline{L} \in \mathsf{LExp}^n$，$f : \mathbb{N} \to \mathsf{Val}^n$ に対して，$\mathsf{array}(\overline{L}, l, f)$ や $\mathsf{sarray}(\overline{L}, s, l, f, d, i)$ を図 2.10 の定義で $L[j] \mapsto f(j)$ の代わりに $\overline{L}[j] \mapsto f(j)$ を用いることで定義する．

$$writeArray(\overline{L}, i, \overline{E}) := (L_0[i] := E_0; L_1[i] := E_1; \ldots; L_{n-1}[i] := E_{n-1})$$
$$readTuple(\overline{x}, \overline{E}) := (x_0 := E_0; x_1 := E_1; \ldots; x_{n-1} := E_{n-1})$$

図 4.1: 配列アクセスコードを生成するメタ言語関数の定義

4.2　配列アクセスのためのメタ言語関数

本研究では配列アクセスコードを生成するメタ言語関数として，タプル配列への書き込み $writeArray(\overline{L}, i, \overline{E})$，タプル式の変数への代入 $readTuple(\overline{x}, \overline{E})$ を考える (図 4.1)．これらの関数は引数のリストの長さが全て等しいときのみ定義される．

以下ではこれらの関数について成り立つ性質について議論する．$writeArray(\overline{x}, \overline{L}, i)$ について，以下の性質が成り立つ．

補題 21 (*writeArray*)**.** 任意の $\overline{L}$，i，$\overline{E}$ について，$\#\overline{L} = \#\overline{E}$ ならば以下が成り立つ．

1. $\mathsf{barriers}(writeArray(\overline{L}, i, \overline{E})) = \emptyset$
2. $\mathsf{writes}(writeArray(\overline{L}, i, \overline{E})) = \emptyset$
3. $\{\overline{L}[i] \mapsto -\}\ writeArray(\overline{L}, i, \overline{E})\ \{\overline{L}[i] \mapsto \overline{E}\}$

$\mathsf{barriers}(C)$，$\mathsf{writes}(C)$ はそれぞれ C 中に出現するバリアの添字の集合，C 中で書き込まれる変数の集合を表す．この補題は $\overline{L}$ の長さに関する帰納法で容易に示すことができる．これらの性質は $writeArray(\overline{L}, i, \overline{E})$ を用いるプログラムを $L[i] := E$ を用いるプログラムと同じように検証できることを意味する．$readTuple(\overline{x}, \overline{E})$ についても同様の性質が成り立つ．

補題 22 (*readTuple*)**.** 任意の $\overline{x}$，$\overline{E}$，$\overline{v}$ について，$\#\overline{x} = \#\overline{E} = \#\overline{v}$ ならば以下が成り立つ．

1. $\mathsf{barriers}(readTuple(\overline{x}, \overline{E})) = \emptyset$
2. $\mathsf{writes}(readTuple(\overline{x}, \overline{E})) = \overline{x}$
3. $\mathsf{disjoint}(\overline{x}), (\bigcup_i \mathrm{fv}(E_i)) \cap \overline{x} = \emptyset$ ならば $\{\overline{E} = \overline{v}\}\ readTuple(\overline{x}, \overline{E})\ \{\overline{x} = \overline{v}\}$

ここで $\mathsf{disjoint}(\overline{x}) := \forall i\, j. i \neq j \Rightarrow x_i \neq x_j$ であり，$\mathrm{fv}(E)$ は E 中に出現する変数を表す．

4.3　ユーザ関数コード

GPUDSL のユーザ関数はタプル値を引数を取り，算術式，条件分岐，自由変数を通じた配列の読み込み等を行うラムダ式である．以下では 1 引数ユーザ関数のみを考える (2 引数以上のユーザ関数も同様に議論できる)．本研究では具体的なユーザ関数の構文規則については議論せず，その表示 $\Downarrow_f$: $\mathsf{Val}^n \times \mathsf{Val}^m \to \mathsf{Prop}$ を用いて議論する．$\overline{v} \Downarrow_f \overline{v'}$ は，ユーザ関数を $\overline{v}$ に適用した時の評価結果が $\overline{v'}$ であることを意味する．

ユーザ関数をコンパイルした結果をテンプレートの穴に直接埋め込むため，ユーザ関数コードは式のタプルを 1 つ受け取って，ユーザ関数の計算を行うコマンドと結果の式のリストの組を返すメタ論理の関数 ($\mathsf{Exp}^n \to \mathsf{Cmd} \times \mathsf{Exp}^m$) として表現される．例えばユーザ関数 $\lambda x.\mathbf{if}\ x \geq 0\ \mathbf{then}\ x\ \mathbf{else}\ -x$ に対応するユーザ関数コードは，$\lambda E_x.((\mathbf{if}\ E_x \geq 0\ \mathbf{then}\ \mathtt{l} := E_x\ \mathbf{else}\ \mathtt{l} := -E_x), \mathtt{l})$ となる．本稿ではユーザ関数コード f に対して，$f(\overline{E})$ の第一要素を $f_c(\overline{E})$，第二要素を $f_e(\overline{E})$ と書く．ユーザ関数コードの仕様は以下のように記述される．

定義 23 (1 引数ユーザ関数コードの仕様 (forward simulation))**.** 接頭辞 "l" を持つ変数が現れない環境 env，ユーザ関数コード $f : \mathsf{Exp}^n \to \mathsf{Cmd} \times \mathsf{Exp}^m$ 及びユーザ関数の表示 $\Downarrow_f$: $\mathsf{Val}^n \times \mathsf{Val}^m \to \mathsf{Prop}$ について以下が成り立つとき，f は env のもとで $\Downarrow_f$ を計算するという

1. 任意の $y \in \mathsf{writes}(f_c(\overline{x}))$ について，y は接頭辞 "l" をもつ
2. 任意の $y \in \mathrm{fv}(f_e(\overline{x}))$ について，y は接頭辞 "l" をもつか，または $y \in \overline{x}$

3. $\mathsf{barriers}(f_c(\overline{x})) = \emptyset$
4. 任意の $\overline{x}$，$\overline{v}$，$\overline{v'}$ について，各 x_i が接頭辞 "l" を持たず $\overline{v} \Downarrow_f \overline{v'}$ ならば
   $\{!(\overline{x} = \overline{v}) \star [\![\mathsf{env}]\!]_{1/n_t n_b}\}\ f_c(\overline{x})\ \{!(f_e(\overline{x}) = \overline{v'}) \star [\![\mathsf{env}]\!]_{1/n_t n_b}\}$

条件 1，2 はコードテンプレートとユーザ関数コードの変数の独立性を保証する．本研究では簡易な接頭辞ベースの変数管理を行う GPUDSL コンパイラを対象とする．接頭辞ベースの変数管理では衝突を許さない変数に異なる接頭辞を持たせることで，変数の独立性を保証する．本研究では Accelerate コンパイラを参考に，ユーザ関数コードが用いる全ての変の変数名は接頭辞 "l" を持つことを仮定した．これに加えてコードテンプレートが用いる変数名が接頭辞 "l" をもたないように制限することで，コードテンプレートとユーザ関数コードの変数の独立性を保証できる [1]．

条件 4 は適当な値 $\overline{v}$ が代入された変数 $\overline{x}$ のもとでユーザ関数コード f が表示 $\Downarrow_f$ をもつユーザ関数を計算することを意味する．環境 env はタプル配列の先頭アドレスを示す変数列，配列の長さ，配列に格納されている値を表す関数の 3 つ組のリストであり，$[\![\mathsf{env}]\!]_{1/n_t n_b}$ はそのような配列群にパーミッション $1/n_t n_b$ でアクセスするための言明が分離積で繋がれたものである．形式的な定義は以下のようになる．

$$\mathsf{env} \in \left\{ \left\{ (\overline{L_i}, l_i, f_i) \right\}_{0 \le i < m} \mid m, k_i \in \mathbb{N},\ f_i \in \mathbb{N} \to \mathsf{Val}^{k_i},\ \overline{L_i} \in \mathsf{LExp}^{k_i} \right\}$$

$$[\![\mathsf{env}]\!]_p := \bigstar_{i=0}^{m-1} \mathsf{array}_p(\overline{L_i}, l_i, f_i)$$

4.4　事例研究 1: map コードテンプレートの検証

事例研究としてまず Accelerate の *mkMap* テンプレートを本手法で検証する．*mkMap* テンプレートは map スケルトンに対応するコードテンプレートである．map スケルトンは d_{in} 次元タプル配列の各要素に型 $T^{d_{in}} \to T^{d_{out}}$ をもつ関数を適用し d_{out} 次元のタプルを計算する．以下に *mkMap* テンプレートの定義を示す．

```
1 mkMap(get,func) :=
2   ix := gid;
3   while (ix < len) {
4     get_c(ix);
5     readTuple(xs, get_e(ix));
6     func_c(xs);
7     writeArray(Gl out, ix, func_e(xs));
8     ix := ix + n_t n_b;
9   }
```

コードテンプレート *mkMap* は 2 つのユーザ関数コード $get : \mathsf{Var} \to (\mathsf{Cmd}, \mathsf{Exp}^{d_{in}})$ と $func : \mathsf{Var}^{d_{in}} \to (\mathsf{Cmd}, \mathsf{Exp}^{d_{out}})$ を入力にとる．*mkMap*(*get*,*func*) から生成されるカーネルは *get*(ix) で配列の各要素を読み込み，それに *func* を適用した要素を配列 *out* に格納する．ここで *xs*，*out* は d_{in}，d_{out} 個の変数の列 x0, x1, x2, ...，out0, out1, out2, ... であり，それぞれ $\mathsf{disjoint}(out)$ 及び $\mathsf{disjoint}(xs)$ を満たし，どの変数も接頭辞 "l" を持たない．

mkMap テンプレートは入力配列からの読み込みがユーザ関数コード *get* で抽象化されている．これは，Accelerate などの GPUDSL はスケルトン間の融合変換をサポートしており，入力配列が実際の GPU 上の配列にならない場合があるためである．例えばソース言語のプログラム map $(\lambda x.x+1)$ (generate 10 $(\lambda x.2x)$) に対しては *mkMap*($\lambda E.$(**skip**, $2 * E$), $\lambda E.$(**skip**, $E + 1$)) でカーネルが生成される．ここで generate n f は各要素が $f(0)$ から $f(n-1)$ で初期化された配列を作成するスケルトンである．

mkMap テンプレートの仕様は以下のようになる [2]．

定理 24. 任意の l，env に対して，

[1] 具体的な接頭辞は重要ではなく，単にコードテンプレートとユーザ関数コードの用いる変数の集合が独立であり，変数が与えられたときにどちらに属するかを決定できればよい

[2] ここでも脚注 2 と同様に事後条件に変数が現れないように変形する必要がある．正確な定義は省略するが，事後条件の [![env]!] に関してもその定義を多少変更することで変数が出現しないように書き換えることができる

- *get* は env のもとで $\Downarrow_g$ を計算し，*func* は env のもとで $\Downarrow_f$ を計算し，かつ
- 任意の $i < l$ に対して $\exists v_g.i \Downarrow_g v_g$ であり，かつ任意の v_g に対して $i \Downarrow_g v_g$ ならば $\exists v_f.v_g \Downarrow_f v_f$

とする．このとき，関数 h が存在して，$\forall i < l.\exists v_g.i \Downarrow_g v_g \Downarrow_f h(i)$ を満たし，

$$\{\texttt{len} = l \land \mathsf{env} \star \mathsf{array}(\mathbf{Gl}\ out, -, l)\}\ mkMap(get, func)\ \{\mathsf{env} \star \mathsf{array}(\mathbf{Gl}\ out, h, l)\}$$

が成り立つ．

2 番目の条件でソース言語の意味論でユーザ関数が安全に実行できること，つまり配列長 l に対して l より小さい全ての i で $g(i)$ を評価できることと，その評価結果 g_v に対して $f(g_v)$ が評価できることを仮定している．証明は，まず $\forall i, i < l \Rightarrow \exists v_g.i \Downarrow_g v_g \Downarrow_f h(i)$ を満たす h の存在を 2 番目の条件より示す．その h のもとでスレッド毎の事前条件，事後条件，およびループ不変式を以下のように定める．

$$\begin{aligned}
P_{i,j} :=& \texttt{len} = l \land [\![\mathsf{env}]\!]_{n_t n_b} \star \mathsf{sarray}(\mathbf{Gl}\ out, 0, l, -, n_t n_g, i + n_t j) \\
Q_{i,j} :=& [\![\mathsf{env}]\!]_{n_t n_b} \star \mathsf{sarray}(\mathbf{Gl}\ out, 0, l, h, n_t n_g, i + n_t j) \\
I :=& \exists x.\texttt{len} = l \land \texttt{ix} = x \cdot n_t n_g + i + n_t j \land [\![\mathsf{env}]\!]_{n_t n_b} \star \\
& \mathsf{sarray}(\mathbf{Gl}\ out, 0, l, \lambda k.\mathbf{if}\ k < x \cdot n_t n_g + i + n_t j\ \mathbf{then}\ h(k)\ \mathbf{else}\ -, n_t n_g, i + n_t j)
\end{aligned}$$

これらの言明のもとでスレッド毎の実行の証明は 3 節の map カーネルの検証とほぼ同様にできる．ユーザ関数に要求した変数に関する条件を Frame 規則の適用時に用いる点が map カーネルの検証とは異なる．

4.5　事例研究 2: reduce テンプレートの検証

2 つ目の事例研究として，GPU のブロック内のスレッド数の最大値を用いてループ展開を行うような Accelerate の *reduceTreeBlock* (以下単に *reduce* と書く) コードテンプレートを検証する．以下は *reduce* テンプレートの定義である [3]．

```
1 reduceBody_n(s) := (
2   barrier_{2n-2};
3   if (tid + s < len) {
4     x0 := Sh sdata[tid + s]; x2 := x1 + x0; x1 := x2
5   }
6   barrier_{2n-1};
7   Sh sdata[tid] = x1)
8 reduce := (reduceBody_1(2^{e_b-1}); ... reduceBody_n(2^{e_b-n}); ... reduceBody_{e_b}(1))
```

ここでは単純のためタプル配列アクセスやユーザ関数コードは排除し，整数の配列を足し算で畳み込むテンプレートを考える．本稿の執筆時点では証明は完了していないが，一般の場合の *reduce* テンプレートで導入されるメタ言語コードについても *mkMap* テンプレートと同様に扱うことができると考えている．

reduce テンプレートは共有メモリ `sdata` の 0 番目から `len` – 1 番目までの要素をブロック内で畳み込むカーネルを生成する．`len` はブロック数 n_t 以下であることを要求する．`sdata` は長さ n_t の配列であり，`len` よりも大きい位置にある要素は無視される．各スレッドは自分のスレッド ID から幅 s だけ右に離れた要素にアクセスし，自分のスレッド ID の要素と足し合わせ共有メモリにそれを書きこむ．ブロック全体としてはループの本体を実行するたびに `len` – s 個の要素の組を足し合わせる．配列のアクセスを行う前後でバリア同期をとることで競合状態を避けている．2^{e_b-1} から $2^0(= 1)$ まで *reduceBody* を動作させることで `sdata` 内の `len` 個の要素を畳み込むことができる．ここで 2^{e_b} は GPU の世代ごとに決められたブロック内のスレッド数 n_t の最大値であり，$2^{e_b} \geq n_t$ である．

reduce テンプレートは実際のカーネルの一部 (ブロック内の配列の畳込み) のみを生成するテンプレートであるが，Grid 規則や Block 規則はカーネル全体にしか適用できない．よってここでは各スレッド単

[3] 実際に Accelerate で用いられる *reduce* テンプレートは GPU が Warp と呼ばれる単位で SIMD 実行を行うことを仮定した最適化 (warp-synchronous programming) を行っているが，本研究ではそのような最適化を行わないカーネルを示す．warp-synchronous programming への対応については 6 章で議論する

位の実行の検証とバリア仕様の正しさの検証のみをおこなう．*reduce* テンプレートの仕様は以下のようになる．

定理 25 (*reduce* テンプレート)**.**

$$\forall l\ f.l \le n_b \Rightarrow \exists f'.$$
$$\mathsf{BS}, i \vdash_T \quad \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f(i) \wedge \mathtt{x1} = f(i) \wedge \mathtt{len} = l\}$$
$$\mathit{reduce}$$
$$\left\{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f'(i) \wedge f'(0) = \sum_{i=0}^{l} f(0)\right\}$$

BS の定義は後であたえる．この仕様は *reduce* が停止するとき，sdata の 0 番目の要素には配列 sdata の初期値の 0 番目から $l-1$ 番目の和が代入されることを意味する．

reduce テンプレートの検証のために，simulating function を用いた証明手法を提案する．*reduce* テンプレートを検証するために，simulating function $f_n(i)$ を導入する．Simulating function とは，カーネルの実行を模倣するようなメタ論理の関数であり，ここでは $f_n(i)$ は n ステップ目における i 番目の配列要素を表す関数である．関数 $f_n(i)$ は以下のように定義される．

$$s_n \coloneqq 2^{e_b - n}$$
$$f_0(i) \coloneqq f(i)$$
$$f_{n+1}(i) \coloneqq \begin{cases} f_n(i + s_{n+1}) + f_n(i) & (i + s_{n+1} < l), \\ f_n(n) & (\text{otherwise}). \end{cases}$$

s_n は n ステップ目の s の値を表す．$f_{n+1}(i)$ は $\mathit{reduceBody}_{n+1}(2^{e_b-(n+1)})$ が sdata に対して行う計算と同じ計算を f_n に対して行う．

Simulating function を用いたカーネルの証明は (i) simulating function が期待される計算を行うこと，及び (ii) カーネルが simulating function と同じ計算を行うことの 2 つを示すことで行われる．まず (i) を示すために，以下の補題を示す．

補題 26.

$$\sum_{i=0}^{\min(l, s_{n+1})-1} f_{n+1}(i) = \sum_{i=0}^{\min(l, s_n)-1} f_n(i)$$

これは $l \le s_{n+1}$，$s_n < l < s_{n+1}$ 及び $l \le s_n$ で場合分けすれば示せる．

証明. $l \le s_{n+1}$ のときは定義より $f_{n+1}(i) = f_n(i)$，$l \le s_{n+1} < s_n$ なので明らか．$l > s_{n+1}$ とする．

$$\begin{aligned}(\text{左辺}) &= \sum_{i=0}^{s_{n+1}-1} f_{n+1}(i) \\ &= \sum_{i=0}^{\min(s_{n+1}-1, l-s_{n+1}-1)} (f_n(i) + f_n(i + s_{n+1})) + \sum_{i=l-s_{n+1}}^{s_{n+1}-1} f_n(i) \\ &= \sum_{i=0}^{\min(s_{n+1}, l-s_{n+1})-1} f_n(i) + \sum_{i=l-s_{n+1}}^{s_{n+1}-1} f_n(i) + \sum_{i=s_{n+1}}^{\min(s_{n+1}, l-s_{n+1})+s_{n+1}-1} f_n(i)\end{aligned}$$

$l - s_{n+1} \le s_{n+1}$ のときは $l \le s_n$，$s_{n+1} + s_{n+1} = s_n$ を用いれば示せる．$l - s_{n+1} > s_{n+1}$ とする．このとき $2s_{n+1} = 2s_n$ より $s_n < l$

$$= \sum_{i=0}^{s_{n+1}-1} f_n(i) + \sum_{i=l-s_{n+1}}^{s_{n+1}-1} f_n(i) + \sum_{i=s_{n+1}}^{2s_{n+1}-1} f_n(i)$$

$$= \sum_{i=0}^{2s_{n+1}-1} f_n(i) = \sum_{i=0}^{\min(l,s_n)-1} f_n(i)$$

□

補題 26 より $\forall n. \sum_{i=0}^{\min(l,s_n)-1} f_n(i) = \sum_{i=0}^{\min(l,s_0)-1} f_0(i)$ であるが，$s_0 = 2^{e_b} \geq n_b \geq l$ より $\sum_{i=0}^{\min(l,s_n)-1} f_n(i) = \sum_{i=0}^{l-1} f(i)$．ここで $n = e_b$ とすると次が成り立つ．

系 27 (simulating function の計算)**.**

$$f_{e_b}(0) = \sum_{i=0}^{l} f(i)$$

これは $f_n(i)$ が *reduce* で期待される畳み込みを行うことを表す．
次に (ii) を示す．BS を以下のように定義する．

$$\mathsf{BS}(i, 2n-2) \coloneqq (\textbf{Sh}\ \texttt{sdata}[i] \mapsto^1 f_n(i), \mathsf{array}_{1/n_b}(\textbf{Sh}\ \texttt{sdata}, n_b, f_n))$$
$$\mathsf{BS}(i, 2n-1) \coloneqq (\mathsf{array}_{1/n_b}(\textbf{Sh}\ \texttt{sdata}, n_b, -), \textbf{Sh}\ \texttt{sdata}[i] \mapsto^1 -)$$

BS の副条件である $\forall 1 \leq n \leq e_b, t \in \{1,2\}. \star_i \mathsf{BS}_{pre}(i, 2n-t) \Rightarrow \star_i \mathsf{BS}_{post}(i, 2n-t)$ は容易に示せる．(ii) は以下の補題として表現される．

補題 28 (simulating function とカーネルの関係)**.**

$$\mathsf{BS}, i \vdash_T \{\textbf{Sh}\ \texttt{sdata}[i] \mapsto f_n(i) \land \texttt{x1} = f_n(i)\} \mathit{reduceBody}_{n+1}(s_{n+1}) \{\textbf{Sh}\ \texttt{sdata}[i] \mapsto f_{n+1}(i) \land \texttt{x1} = f_{n+1}(i)\}$$

これは $f_{n+1}(i)$ と $\mathit{reduceBody}_{n+1}(s)$ の定義から容易に示せる．
補題 28 と補題 26 を用いると定理 25 は以下のように示せる．

証明. *reduce* は $\mathit{reduceBody}_1(s_1); \ldots \mathit{reduceBody}_i(s_i); \ldots \mathit{reduceBody}_{e_b}(s_{e_b})$ と書けることから，補題 28 より

$$\mathsf{BS}, i \vdash_T \{\textbf{Sh}\ \texttt{sdata}[i] \mapsto f_0(i) \land \texttt{x1} = f_0(i)\} \mathit{reduce} \{\textbf{Sh}\ \texttt{sdata}[i] \mapsto f_{e_b}(i) \land \texttt{x1} = f_{e_b}(i)\}$$

f' として f_{e_b} をとれば，系 27 より定理の結果を得る．□

さらに *reduce* テンプレートを以下のように変形することでバリア同期命令を 1 つ省略することができる．

```
1 reduceBody_n(s) := (
2   barrier_n
3   if (tid + s < len && tid < s) {
4     x0 := Sh sdata[tid + s]; x2 := x1 + x0; x1 := x2; Sh sdata[tid] = x1
5   })
6 reduce := (reduceBody_1(2^(e_b-1)); ... reduceBody_n(2^(e_b-n)); ... reduceBody_{e_b}(1))
```

バリア同期の省略の他に，条件式に $\texttt{tid} < s$ が加わり，共有メモリへの書込みが条件分岐の中に移動した点が異なる．この最適化の正しさの証明の詳細は割愛するが，simulating function を $i + s < l \land i < s$ で場合分けするように変更し，バリア仕様は書き込みを行うスレッド i が i と $i + s$ に対して排他的なパーミッションを持つように変更すれば良い．以下に simulating function とバリア仕様の定義を示す．

$$f_0(i) \coloneqq f(i)$$
$$f_{n+1}(i) \coloneqq \begin{cases} f_n(i + s_{n+1}) + f_n(i) & (i + s_{n+1} < l \land i < s_{n+1}), \\ f_n(n) & (\text{otherwise}). \end{cases}$$
$$\mathsf{BS}(i, n)_{pre} \coloneqq \begin{cases} P(f_{n-1}, n-1, i) & (n > 1) \\ \textbf{Sh}\ \texttt{sdata}[i] \mapsto^1 f(i) & (n = 1) \end{cases}$$

$$\mathsf{BS}(i,n)_{post} := P(f_{n-1},n,i)$$

$$P(f,n,i) := \begin{cases} \mathbf{Sh}\ \mathtt{sdata}[i] \mapsto^1 f(i) \star \mathbf{Sh}\ \mathtt{sdata}[i+s_n] \mapsto^1 f(i+s_n) \star & \\ \quad \mathbf{if}\ i=0\ \mathbf{then}\ R_n\ \mathbf{else}\ \mathbf{emp} & (i+s_n < l \wedge i < s_n) \\ \mathbf{emp} & (\text{otherwise}) \end{cases}$$

R_n の定義は省略するが，n ステップ目の実行でアクセスされない位置 $(\min(l-s_n,s_n) \le i < s_n$ または $\min(l-s_n,s_n)+s_n \le i < n_t$ を満たす $i)$ を表すリソースである．BS の副条件は $\forall n,f.\ \bigstar_{i\in\mathsf{Tid}} P(f,n,i) \Leftrightarrow \mathsf{array}(\mathbf{Sh}\ \mathtt{sdata},n_t,f)$ から容易に示せる．この simulating function とバリア仕様のもとで以下の定理が定理 25 と同様に示せる．

定理 29 (*reduce* テンプレート)**.**

$$\forall l\ f.l \le n_b \Rightarrow \exists f'.$$

$$\mathsf{BS}, i \vdash_T \quad \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f(i) \wedge \mathtt{x1} = f(i) \wedge \mathtt{l} = l\}$$

$$\textit{reduce}$$

$$\left\{\mathbf{if}\ i=0\ \mathbf{then}\ \mathsf{array}(\mathbf{Sh}\ \mathtt{sdata},l,f') \wedge f'(0) = \sum_{i=0}^{l-1} f(i)\ \mathbf{else}\ \mathbf{emp}\right\}$$

第5章　関連研究

5.1　GPGPU のための自動検証器

GPGPU カーネルの競合状態やバリア相違を検出する検証器として Betts らによる GPUVerify [8, 4, 9] や Li らによる GKLEE [25, 26] がある．

Betts らは競合状態やバリア相違を表現することができる意味論である SDV 意味論を設計し，その意味論のもとで健全な検証器 GPUVerify を設計・実装した．SDV 意味論は各スレッドが 1 ステップずつ同期を取りながらプログラムを実行するような意味論であり各スレッドが読み込み・書き込みを行ったアドレスの集合 (read/write set) を記録する．バリア同期時やカーネルの停止時にこれまでの read set と write set，write set と write set の間に重複があったときに競合を起こしているとみなされる．自動検証の際に証明義務に全称量化が出現するのを避けるために，GPUVerify はこれらの性質を任意の 2 スレッドの間での性質に帰着させる手法を用いている (two threads reduction)．GPUVerify は two threads reduction を用いて GPU カーネルを逐次プログラム検証のための検証器 Boogie [5] の対象言語に変換し，Boogie 上で検証を行う．`assert` 文と `assume` 文を用いることで，GPUCSL と同じように仕様検証を行うことができるが，証明できる仕様は，GPUVerify が利用する SMT ソルバで解ける仕様に制限される．

Li らは concolic execution に基づく GPU カーネルのための検証・テスト生成器である GKLEE を提案した．Concolic execution と一は部の入力に対して具体値を用いるような記号的実行である．任意のスケジューリングに対して記号実行を行うには膨大な計算量が必要となるため，GKLEE は canonical scheduling というスケジューリングのみに対して記号実行を行う．Canonical scheduling のもとでエラーを起こさないと検証されたカーネルは任意のスケジューリングのもとでエラーを起こさない．GKLEE は warp divergence や bank conflict といった計算は正しく行われるが，計算速度に影響を与えるようなプログラムの検出も行うことができる．

5.2　プログラム論理

Vafeiadis は CSL の健全性の簡潔な証明を提案し，それを FPCSL の健全性の証明に応用した [36]．また証明を Coq と Isabelle/HOL 上で行った．本研究では Vafeiadis の証明法を GPUCSL の健全性証明に応用した．

Blom らは GPGPU のための CSL を提案し，その論理に基づく自動検証器を作成した [10]．本研究では Blom らの CSL を拡張することでコードテンプレートの検証を行った．

小島らは single instruction multiple threads (SIMT) プログラムのための Hoare 論理を提案し，その健全性と相対完全性を証明した [22]．SIMT は全てのスレッドの実行が同時に 1 ステップずつ進む意味論である．本研究では各スレッドの実行は任意の順序で進むとした．

Affeldt らは TLS パケット処理を行う C 言語プログラムの検証のために Coq/SSReflect [18] 上で分離論理を形式化した [1]．彼らの分離論理はデータ構造のアラインメントや `sizeof` 演算子などの細かい仕様を表現することができる．本研究の対象言語では複合データ型は除外し，データ型は整数のみとした．

Hobor らは Pthreads 風のバリア同期のための並行分離論理を提案した [21]．彼らは対象言語として Compcert コンパイラの中間表現 Cminor を採用し，Coq で健全性を証明した．本研究との違いとして，Hobor らの研究では対象言語でスレッドの動的な作成が許されていることや，バリア相違に対する検証は行っていないことが挙げられる．

5.3　GPGPU の意味論

CUDA C はメモリアクセスの順序が全スレッドで一貫して観測されない弱いメモリモデルに基づく意味論を持つが，本稿の執筆時点でその形式的な意味論の定義は定められていない．Alglave らはそのような CUDA のメモリモデルの定義を与えるために大量のテストケースを用いて GPU ハードウェアのメモリモデルを調査し，その結果をもとにメモリモデルの定義を提案した [2]．WhileB 言語は逐次一貫性をもつ強いメモリモデルに従うため，本研究の意味論は正確ではないが，競合状態が起こらないプログラムに関しては逐次一貫性を持つと考えられるため，GPUCSL は実際の意味論の上でも健全であると予想される．

5.4　コード生成器の正しさ

GPUDSL は並列スケルトンに対する自動並列化と考えることができるが，自動並列化を行うコンパイラの検証として Bell による研究 [7] がある．健全な自動並列化の議論ために，彼らは Calculsus of Communicating Systems (CCS) のもとで一般的な双模倣関係より弱い模倣関係を定義し，そのもとでいくつかの最適化が健全であることを示している．一方 GPUDSL は副作用を持たないため，観測可能な振る舞いとして評価結果の値しかない．またソース言語は並列スケルトンとユーザ関数しかもたないため，このような複雑な模倣関係の導入は不要であると考えられる．

本研究で対象とした正しさよりも弱い正しさを保証する研究として，ソースコードと生成コードの間で自動的に型保存が行われることが保証されたコンパイラ (型保存コンパイラ) に関する研究が知られており，GPUDSL においても McDonell らによる研究 [28] がある．このようなコンパイラは本研究で対象としたような接頭辞ベースの変数管理よりも複雑な変数管理機構を用いる．そのようなコンパイラの生成コードの計算の正しさを保証することは興味深い問題である．

5.5　安全性と容易性を両立する高性能プログラミング機構

Kolesnicheko らは Eiffel 上で契約を用いて並列スケルトンによる GPGPU プログラミングを行うためのライブラリ SafeGPU を提案した [23]．この研究は生産性が高く安全なプログラムを記述できるライブラリの提供という動機を持つ．つまり，並列スケルトンを用いて簡単に高性能な GPGPU プログラムを書くことができると同時に，Eiffel の契約機構を用いることでプログラムの安全性や正しさの保証も高めることができる．SafeGPU は本研究と似た動機をもつが，彼らの研究ではコンパイラそのものの正しさについては扱っていない．

江本らは GTA フレームワークの Coq による検証を行った [17]．GTA フレームワークはプログラムを解候補の生成，検査と半環による集約の合成で記述し，半環融合変換 (semi-ring fusion) を用いることで，記述されたプログラムからより高速な並列プログラムを導出する枠組みである．彼らは GTA フレームワークを Coq 上で形式化し，形式化のもとで変換の正しさを証明した．また，Coq 上で証明されたプログラムから実際に動作する BSML [27] プログラムへの抽出を示した．BSML は bulk synchronous parallel モデルに基づく OCaml ライブラリであり，MPI といった分散メモリ並列ライブラリの上で動作する．本研究との違いとして，本研究では共有メモリ並列である GPGPU へのコード生成を対象としている点や，江本らの研究では行っていない生成されたコードの検証を行っている点が挙げられる．一方で GTA フレームワークを本研究によって検証されたコード生成器に用いることで，生成コードのレベルまで検証された高速な GPGPU プログラムの抽出が可能になると考えている．

数学的に記述された仕様から高性能なプログラムを演繹的に合成する機構として，Delaware らによる Coq ライブラリ Fiat [16] がある．Fiat を用いるユーザは高水準な言語で記述されたプログラム仕様から，実際に実行可能な OCaml プログラムを Coq のタクティックを用いることで対話的に導出する．Fiat の自動生成タクティックを用いることで，プログラムの導出は自動的に行うこともできる．導出の各過程は Coq の定理として記述されており，導出されたプログラムはもとの仕様に従って動作することが保証さ

れる．Fiat は高水準な言語で記述されたプログラムから，それと同じ振る舞いを示すことが保証された高速なプログラムを生成するという点で本研究と似た動機を持つ．しかし，本研究では並列プログラム (GPGPU コード) の生成を行うコンパイラを対象としていて，Fiat は逐次プログラム (OCaml コード) の生成を行っている．また，本研究で対象とした GPUDSL は Fiat のようにユーザが対話的に高速なプログラムの導出を行うことは想定していない．

第6章　結論・今後の課題

本研究では並行分離論理と定理証明支援器を用いて GPGPU コードテンプレートを検証する手法を提案した．以下に本研究の貢献を述べる．

- 一般の Frame 規則で拡張された GPUCSL の健全性の形式証明を初めて行い，Blom らによる GPUDSL の健全性の証明にあった証明の不備を明らかにした．Frame 規則はコードテンプレートだけでなく，関数呼び出しを含むカーネルに対してモジュラーに検証を行うために必要な規則であるため，Frame 規則で拡張された GPUCSL の健全性の証明は重要である．また，Blom らが与えなかったスレッド ID 独立性の健全性の証明を初めて示し，彼らが言及しなかった健全性のための条件を明らかにした．
- GPGPU プログラム検証のための Coq ライブラリ GPUVeLib を作成し，map，reduce，prescan といったカーネルに対する検証を行った．これまで逐次プログラムや fork-join 並列モデルに基づく並行プログラムに対する検証は多く行われてきたが，GPGPU のようなデータ並列モデルに基づくプログラムの計算の正しさの検証を行った事例は我々の知る限り存在しない．GPUVeLib は GPUCSL に基づくカーネル検証の中で最も難しい点である，各スレッドの逐次実行に関する検証を補助するためのライブラリである．本研究で検証した reduce や prescan カーネルは GPGPU において重要な基礎要素であるため，これらのカーネルの計算の正しさを含めた検証は有用である．
- GPGPU コードテンプレートの正しさについての議論を行い，特に正しいコードを生成するためにユーザ関数コードに要求される条件を明らかにした．これはユーザ関数コードの用いる変数の独立性やユーザ関数の意味論との関係といった条件を含む．これに基づいて Accelerate の map，reduce テンプレートに対して検証を行った．その結果，一般のコードテンプレートに対する本手法の適用可能性を確認することができた．
- Reduce コードテンプレートのような複雑なメモリアクセスパターンをもつテンプレートに対して simulating function を用いる検証手法を提案した．Simulating function を用いて Accelerate の reduce テンプレートの検証を検証を行い，その過程で最適化の余地を発見し，最適化されたテンプレートの正しさを検証した．

本研究の今後の課題として，現実的な規模の GPUDSL に本研究を適用することが考えられるが，以下ではその際に問題となりうる課題について述べる．

6.1　コンパイラ全体の検証

今後の課題として，GPUDSL コンパイラ全体，つまりソース言語から CUDA コードの変換を検証することが挙げられる．そのためにはコンパイラフロントエンドで行われる融合変換やユーザ関数のコンパイルを検証する必要がある．例えば，並列スケルトンに対する融合変換は McDonnel ら [29] によって議論されているる．これらの融合変換の正しさは直感的には明らかであるが，融合変換を行うコンパイラの実装を含めた検証は興味深い問題である．また，ユーザ関数のコンパイルは GPUCSL のスレッド実行の 3 つ組 $(\mathsf{BS}, i \vdash_T \{P\}\ C\ \{Q\})$ の上で正しいことを示す必要がある．これは CompCert [24] などのコンパイラ検証で行われてきたコンパイラの正しさ (ソースプログラムと出力プログラムの間の双模倣性) とは異なる形の仕様であり，証明には若干の工夫を要すると考えられる．

6.2　より高度な CUDA の意味論への対応

本研究で用いた並行分離論理はスレッドブロック，グリッドや共有メモリといった基本的な CUDA の機能をサポートしている．一方でいくつかのコードテンプレートはより高度な CUDA の機能を用いて記述されている．その中には atomic 命令，warp-synchronous programming，warp shuffle 命令といった機能が含まれる．GPGPU 向け並行分離論理で atomic 命令を含むプログラムの検証は Amighi らによって議論されており [3]，我々の手法でも容易に利用できるはずである．

Warp-synchronous programming とは warp 内のスレッドが 1 命令毎に同期的に実行されることを利用してバリア同期を省略するプログラミング手法である．Warp とはスレッド ID を 32 で割った商が等しく，かつ同じスレッドブロックに属するスレッドの集合である．Accelerate コンパイラでも Warp-synchronous programming を用いて reduce テンプレートの最適化を行っている．GPUCSL ではこれらの実行をモデル化していないため，warp-synchronous programming を用いたコードは検証できないが，Nvidia 社の文書ではこの技術は将来の CUDA でも利用できる保証はなく，後述の warp shuffle 命令を用いることを推奨している [32]．そのためこの機能を用いるプログラムを正しいとみなすかどうかは議論の余地がある．このような実行モデルに従うプログラムの検証は，小島らによる SIMT プログラムのための Hoare 論理 [22] を用いることで行うことができると考えられる．

Warp shuffle 命令は共有メモリやグローバルメモリを使わずに warp 内でデータの交換を行うための命令である．Warp shuffle 命令を呼び出した warp 内のスレッド間で引数に指定した値の交換が行われる．GPUCSL の視点から見ると，warp shuffle 命令は warp 単位のリソース (ここでは値) の交換とみなすことができ，バリア同期命令に対するバリア仕様のように warp shuffle 命令に仕様を与えることで検証できると考えられる．

6.3　コードテンプレートの証明の容易化

本研究ではコード検証のためのライブラリを作成し，それを用いてコードテンプレートの検証を行った．GPUVeLib によるコード検証は冗長なスクリプトの記述が多く，まだ多くの改善の余地がある．その中には逐次実行に関する $\mathsf{BS}, i \vdash_T \{P\}\, C\, \{Q\}$ の証明や，リソースの集約に関する条件 ($\bigstar_{i \in \mathsf{Tid}} BS(i,b)_{pre} \Rightarrow \bigstar_{i \in \mathsf{Tid}} BS(i,b)_{post}$ など) の証明がある．逐次実行に関する証明は Chlipala や [14] や Cao ら [12] による証明自動化手法を用いることで同様に自動化できると考えられる．一方でリソースの集約に関する条件は $\bigstar_{i \in \mathsf{Tid}} BS(i,b)$ のように一般的な個数の分離積を扱う必要があり，このような条件の自動的な証明はこれらの自動証明ライブラリでは議論されていない．このような条件の一般的な証明は困難であるが，*mkMap* や *reduce* テンプレートに現れた GPGPU カーネル検証で頻繁に出現するようなパターンに限れば自動化は可能であると考えられる．

また GPGPU プログラムの自動検証技術は Betts らによる GPUVerify [8] など多く存在し，それらはカーネルの競合状態やバリア相違が起こらないこといった一般的な性質を検証する．これらの検証器をコードテンプレートに応用することで，生成される任意のカーネルが競合状態を起こさないことやバリア相違を起こさないことなどを容易に検証できる可能性がある．

参考文献

[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] Alglave, J., Batty, M., Donaldson, A. F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T. and Wickerson, J.: GPU Concurrency: Weak Behaviours and Programming Assumptions, Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pp. 577–591 (2015).

[3] Amighi, A., Darabi, S., Blom, S. and Huisman, M.: Specification and Verification of Atomic Operations in GPGPU Programs, Software Engineering and Formal Methods, Lecture Notes in Computer Science, Vol. 9276, pp. 69–83 (2015).

[4] Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson, A. F., Ketema, J., Liew, D. and Qadeer, S.: Engineering a Static Verification Tool for GPU Kernels, Computer Aided Verification - 26th International Conference, CAV 2014, pp. 226–242 (2014).

[5] Barnett, M., Chang, B. E., DeLine, R., Jacobs, B. and Leino, K. R. M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs, Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, pp. 364–387 (2005).

[6] Barthe, G.(ed.): Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, Lecture Notes in Computer Science, Vol. 6602, Springer (2011).

[7] Bell, C. J.: Certifiably Sound Parallelizing Transformations, Certified Programs and Proofs - Third International Conference, CPP 2013, pp. 227–242 (2013).

[8] 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).

[9] Betts, A., Chong, N., Donaldson, A. F., Ketema, J., Qadeer, S., Thomson, P. and Wickerson, J.: The Design and Implementation of a Verification Technique for GPU Kernels, ACM Trans. Program. Lang. Syst., Vol. 37, No. 3, p. 10 (2015).

[10] 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).

[11] Bornat, R., Calcagno, C., O'Hearn, P. W. and Parkinson, M. J.: Permission accounting in separation logic, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 259–270 (2005).

[12] Cao, J., Fu, M. and Feng, X.: Practical Tactics for Verifying C Programs in Coq, Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, pp. 97–108 (2015).

[13] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L. and Grover, V.: Accelerating Haskell Array Codes with Multicore GPUs, Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, pp. 3–14 (2011).

[14] Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic, Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 234–245 (2011).

[15] Chong, N., Donaldson, A. F., Kelly, P. H. J., Ketema, J. and Qadeer, S.: Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, pp. 605–622 (2013).

[16] Delaware, B., Pit-Claudel, C., Gross, J. and Chlipala, A.: Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, pp. 689–700 (2015).

[17] Emoto, K., Loulergue, F. and Tesson, J.: A Verified Generate-Test-Aggregate Coq Library for Parallel Programs Extraction, Interactive Theorem Proving - 5th International Conference, ITP 2014, pp. 258–274 (2014).

[18] Gonthier, G., Mahboubi, A. and Tassi, E.: A Small Scale Reflection Extension for the Coq system (2015).

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

[20] Harris, M., Sengupta, S. and Owens, J. D.: Parallel Prefix Sum (Scan) with CUDA, GPU Gems, Vol. 3, No. 39, Addison Wesley, pp. 851–876 (2007).

[21] Hobor, A. and Gherghina, C.: Barriers in Concurrent Separation Logic, In Barthe [6], pp. 276–296.

[22] 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).

[23] Kolesnichenko, A., Poskitt, C. M., Nanz, S. and Meyer, B.: Contract-based general-purpose GPU programming, Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, pp. 75–84 (2015).

[24] Leroy, X.: A formally verified compiler back-end, Journal of Automated Reasoning, Vol. 43, No. 4, pp. 363–446 (2009).

[25] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I. and Rajan, S. P.: GKLEE: Concolic Verification and Test Generation for GPUs, Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, pp. 215–224 (2012).

[26] Li, P., Li, G. and Gopalakrishnan, G.: Practical Symbolic Race Checking of GPU Programs, International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, pp. 179–190 (2014).

[27] Loulergue, F., Hains, G. and Foisy, C.: A calculus of functional BSP programs, Sci. Comput. Program., Vol. 37, No. 1-3, pp. 253–277 (2000).

[28] McDonell, T. L., Chakravarty, M. M. T., Grover, V. and Newton, R. R.: Type-safe runtime code generation: accelerate to LLVM, Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, pp. 201–212 (2015).

[29] McDonell, T. L., Chakravarty, M. M. T., Keller, G. and Lippmeier, B.: Optimising purely functional GPU programs, ACM SIGPLAN International Conference on Functional Programming, ICFP'13, pp. 49–60 (2013).

[30] 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.

[31] NVIDIA: CUDA C Programming Guide (2015).

[32] NVIDIA: TUNING CUDA APPLICATIONS FOR KEPLER (2015).

[33] O'Hearn, P. W.: Resources, Concurrency and Local Reasoning, CONCUR 2004 - Concurrency Theory, 15th International Conference, Lecture Notes in Computer Science, Vol. 3170, pp. 49–67 (2004).

[34] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures, 17th IEEE Symposium on Logic in Computer Science (LICS 2002), pp. 55–74 (2002).

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

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

[37] Vafeiadis, V.: Concurrent Separation Logic Soundness, `https://www.mpi-sws.org/~viktor/cslsound/` (2011).