GPGPU向けデータ並列コードテンプレートの形式検証

朝倉 泉[1]，増原 英彦[2]，青谷 知幸[3]

東京工業大学

[1] asakura.i.aa@m.titech.ac.jp [2] masuhara@acm.org [3] aotani@is.titech.ac.jp

概 要 GPGPU 向けデータ並列 DSL (GPUDSL) は map や reduce といったよく知られた並列スケルトンを用いて高性能な GPGPU プログラムを容易に記述するための言語である．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 の並列実行モデルや，(ii) テンプレートの引数に対して生成コードや生成コードの満たすべき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 GPGPU 向け並行分離論理を用いて Coq 上で検証する手法を提案する．さらにこの手法の適用例として map や reduce スケルトンに対応するコードテンプレートの検証も示す．

1 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 [1] 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 [2, 3]．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が生成するプログラムの実行の正しさを保証するための難しさの要因として (i) GPGPU の並列実行モデルそのものに由来する難しさと (ii) コードテンプレートの引数に応じて生成されるコードやその仕様が変化する点がある．(i) の問題を扱う GPGPU コードのための検証技術は多く存在するが [4, 5, 6]，我々の知る限り，(i) と (ii) を同時に扱うことができる検証技術は存在しない．

本研究では定理証明支援器 Coq 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．生成されたコードの正しさを検証するために，GPGPU コードのための並行分離論理 (GPUCSL) [7] をグリッドと共有メモリを扱えるように拡張したものを用いる．また，検証を補助するための Coq ライブラリを作成し，それを用いて GPUDSL の 1 つである Accelerate [2] 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．

また本研究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．我々は 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．

- 先行研究として行った Blom らの GPUCSL を Coq で定式化し健全性を証明した研究に対し，グリッド，共有メモリを扱える拡張を行った
- 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

本稿で示す定義・定理は Coq で形式化されているが，紙面の都合上その形式化の概略のみを示す．

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

2 GPGPU のための並行分離論理 GPUCSL

本研究ではコードテンプレート検証のために，GPGPU プログラムのための並行分離論理 GPUCSL を用いる．これは我々の以前の研究 [7] にグリッドや共有メモリを追加し，Coq で形式化した上で健全性を証明したものである．健全性の証明は先行研究の証明にグリッド規則に関する証明を追加することで行った．本節では主な導出規則について述べ，健全性の証明は省略する．

GPUCSL は CUDA C [1] 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

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

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

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

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

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

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

図 1: WhileB 言語の構文規則

$$v \in \mathsf{Val} := \mathbb{Z}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$L \mapsto^p - := \exists v.L \mapsto^p v$$

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

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

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

図 2: GPUCSL の論理式の意味論

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．

図 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 つのコマンドから成る．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．

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

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

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

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

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

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

図 3: 逐次実行のための証明規則

図 2 は GPUCSL の言明の意味論である．言明は変数環境であるスタック s と共有メモリとグローバルメモリをまとめたヒープ h のもとで評価される．s は変数から値への関数 $s : \mathsf{Var} \to \mathsf{Val}$ であり，h はアドレスからパーミッションと値の組への部分関数 $h : \mathsf{Loc} \rightharpoonup \mathsf{Perm} \times \mathsf{Val}$ である．Loc は共有メモリのアドレスとグローバルメモリのアドレスを区別するタグ **Sh**，**Gl** と整数の組である．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 [8] ($\mathsf{Perm} := \{q \in \mathbb{Q} \mid 0 < q \leq 1\}$) 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．共有メモリとグローバルメモリが存在することを除けば，GPUCSL の言明は通常のパーミッション付き CSL と同じ意味をもつ．

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 $f(j)$ 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．$\mathsf{array}_p(L, m, n, f)$, $\mathsf{sarray}_p(L, m, n, f, d, i)$ は以下の性質を満たす．

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

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

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

図 4: 並列実行のための証明規則

- $(\forall 0 \le j < n, (m+j) \bmod d = i \Rightarrow f_1(m+j) = f_2(m+j)) \Rightarrow$
  $\mathsf{sarray}_p(L, m, n, f_1, d, i) \Leftrightarrow \mathsf{sarray}_p(L, m, n, f_2, d, i)$ (sarray_eq)

本稿では $p = 1$ のときや $m = 0$ のときに $L \mapsto^p E$，$\mathsf{array}_p(L, m, n, f)$ 及び $\mathsf{sarray}_p(L, m, n, f, d, i)$ の p や m を省略して表記する．

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 3 は逐次実行に関する証明規則である．$\mathrm{fv}(E)$ ($\mathrm{fv}(L)$) はそれぞれ E (L) 中に現れる変数の集合，$\mathsf{writes}(C)$ は C の中で書き込み，代入の右辺に出現する変数の集合である．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．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．$\mathsf{BS}(i,b)_{pre}, \mathsf{BS}(i,b)_{post}$ 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，$\star_{i\in\mathsf{Tid}} BS(i,b)_{pre} \Rightarrow \star_{i\in\mathsf{Tid}} BS(i,b)_{post}$ 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 4 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 行目から 7 行目の条件はバリア相違 (barrier divergence) [1] を起こさないことを保証するための規則である．詳しくは我々の以前の研究 [7] を参照されたい．

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，$[\![\overline{s[n]}]\!] = \bigstar_{s_i[n_i]\in\overline{s[n]}} \mathsf{array}(\mathbf{Sh}\ s_i, n_i, -)$ 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，詳細は省略する．

GPUCSL の意味論 $\vDash_G \{P\}\overline{s[n]}; C\{Q\}$ は直感的には，P を満たす状態から $\overline{s[n]}; C$ の実行を開始した

```
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 5: stencil カーネル

とき，

- 実行中に未初期化領域へのアクセスを起こさない，かつ
- バリア相違 (ブロック内の 2 つのスレッドが異なるバリアに到達すること) を起こさない，かつ
- 停止するときのグローバルメモリ状態が Q を満たす

と定義される．GPUCSL の健全性とは $\vdash_G \{P\}\overline{s[n]}; C\{Q\}$ ならば $\models \{P\}\overline{s[n]}; C\{Q\}$ が成り立つことである．健全性の証明は逐次実行，スレッドブロック実行に関する規則は我々の以前の研究で行われており，グリッド実行に関してはスレッドブロック実行に関する証明をグリッド実行に拡張することで行った．本稿では健全性の証明は省略するが，これらは Coq 上で形式化，証明されている．

本稿ではいくつかのプログラムに対して検証の概略を示すが，Block 規則の 5 行目以降，及び Grid 規則の 4 行目以降の条件の証明は省略する．実際，これらの条件の証明の多くの部分は Coq 上で自動化されている．

2.1 例: 1 次元ステンシル

簡単な WhileB プログラムの例を図 5 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．

stencil カーネルの仕様を以下のように与える [1]．

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

ここで $f'(k) := \mathbf{if}\ k < 0 \vee n_t n_b \le k\ \mathbf{then}\ 0\ \mathbf{else}\ f(k)$ 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．

$$P_{i,j} := \boxed{\mathsf{array}_{1/(n_t n_b)}(\mathbf{Gl}\ \mathtt{arr}, n_t n_b, f) \star (\mathbf{Gl}\ \mathtt{out}[i + jn_t] \mapsto^1 -)} \star$$

[1] 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 $\forall f, l_{in}, l_{out}.\ \vdash_G \{\mathtt{arr} = l_{in} \wedge out = l_{out} \wedge \ldots\}\{\ldots\}\{\ldots\}$ 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．

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 1: GPUVeLib の概要

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

$P'_{i,j}$，$Q'_{i,j}$ 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．

- $P \Rightarrow \bigstar_{j\in\mathsf{Bid}} P_j$ (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．

誤ったプログラムを検証する場合の例として図 5 の 10 行目のバリア同期を削除したカーネルを考える．このカーネルは GPUCSL では証明することができない．なぜなら，スレッド i が 3 行目，スレッド $i+1$ が 11 行目の 1 つ目の文を実行するには，それぞれ言明 $\mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^1 -$，$\mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^p -$ (ただし $0 < p \leq 1$) が必要となるが，それらの和 $\mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^1 - \star \mathbf{Sh}\ \mathtt{smem}[i+1] \mapsto^p -$ は事前条件から導くことができないためである．

3 GPGPU カーネル検証のためのタクティックライブラリ 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 上で検証できるが，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．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 {(ix = x * n + tid) ∧ array_{1/n}(in, 0, ix, f) ⋆ Gl in[ix] ↦ f(i) ⋆ ...}
2   t := Gl in[ix]
3 {...}
```

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 2: カーネル検証の概略

これを証明するには (i) 事前条件の **GI** `in[ix]` $\mapsto f(i)$ が分離積 ($\star$) 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 [9] や Chlipala [10] 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．先行研究における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．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．表 1 に GPUVeLib のタクティックの概略を示す．

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)

これらのカーネルの検証の概略を表 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．

4 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 のパラメータなどが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型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あるため，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

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

図 6: 配列アクセスコードを生成するメタ言語関数の定義

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，これを用いることで 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．

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．**Gl** $\overline{E}$ 及び $\overline{L}[i]$ をそれぞれリスト **Gl** E_0, **Gl** $E_1, \ldots$ **Gl** E_{n-1} 及び $L_0[i], L_1[i], \ldots L_{n-1}[i]$ とする (**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 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．

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 6)．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 1 (*writeArray*)**.** $\mathsf{barriers}(C)$，$\mathsf{writes}(C)$ をそれぞれ 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とする．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. $\{\overline{L}[i] \mapsto -\}\ writeArray(\overline{L}, i, \overline{E})\ \{\overline{L}[i] \mapsto \overline{E}\}$

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 $\mathit{writeArray}(\overline{L}, i, \overline{E})$ 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．$\mathit{readTuple}(\overline{x}, \overline{E})$ についても同様の性質が成り立つ．

補題 2 ($\mathit{readTuple}$)**.** $\mathsf{disjoint}(\overline{x}) := \forall ij.i \neq j \Rightarrow x_i \neq x_j$，$\mathrm{fv}(E)$ を E 中に出現する変数とする．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. $\mathsf{barriers}(\mathit{readTuple}(\overline{x}, \overline{E})) = \emptyset$
2. $\mathsf{writes}(\mathit{readTuple}(\overline{x}, \overline{E})) = \overline{x}$
3. $\mathsf{disjoint}(\overline{x}), (\bigcup_i \mathrm{fv}(E_i)) \cap \overline{x} = \emptyset$ ならば $\{\overline{E} = \overline{v}\}\ \mathit{readTuple}(\overline{x}, \overline{E})\ \{\overline{x} = \overline{v}\}$

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に取り，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，$\lambda E_x.((\mathbf{if}\ E_x \geq 0\ \mathbf{then}\ \mathtt{1} := E_x\ \mathbf{else}\ \mathtt{1} := -E_x), \mathtt{1})$ 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 3 (1 引数ユーザ関数コードの仕様)**.** $\mathrm{fv}(E)$ を E 中に現れる変数の集合とする．接頭辞 "1" 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 "1" をもつ
2. 任意の $y \in \mathrm{fv}(f_e(\overline{x}))$ について，y は接頭辞 "1" 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 "1" を持たず $\overline{v} \Downarrow_f \overline{v'}$ ならば
   $\{(\overline{x} = \overline{v}) \wedge [\![\mathsf{env}]\!]_{1/n_t n_b}\}\ f_c(\overline{x})\ \{(f_e(\overline{x}) = \overline{v'}) \wedge [\![\mathsf{env}]\!]_{1/n_t n_b}\}$

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数の変数名は接頭辞 "1" 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 "1" 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．

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．

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

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

```
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   }
```

図 7: *mkMap* テンプレート

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．*mkMap* テンプレートの定義を図 7 に示す．

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 disjoint(*out*) 及び 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 $mkMap(\lambda E.(\mathbf{skip}, 2 * E), \lambda E.(\mathbf{skip}, E + 1))$ 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 4. 任意の l，env に対して，

- *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 $0 \le i < l$ 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，

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

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 のもとで各スレッド i, j の事前条件 $P_{i,j}$，事後条件 $Q_{i,j}$，およびループ不変式 $I_{i,j}$ を以下のように定める．

$$P_{i,j} := \mathtt{len} = l \wedge [\![\mathsf{env}]\!]_{1/n_t n_b} \star \mathsf{sarray}(\mathbf{Gl}\ out, 0, l, -, n_t n_b, i + n_t j)$$

$$Q_{i,j} := [\![\mathsf{env}]\!]_{1/n_t n_b} \star \mathsf{sarray}(\mathbf{Gl}\ out, 0, l, h, n_t n_b, i + n_t j)$$

$$I_{i,j} := \exists x.\mathtt{len} = l \wedge \mathtt{ix} = x \cdot n_t n_b + i + n_t j \wedge [\![\mathsf{env}]\!]_{n_t n_b} \star$$

[2] ここでも脚注 1 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．

```
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))
```

図 8: *reduce* テンプレート

$$\mathsf{sarray}(\mathbf{Gl}\ out, 0, l, \lambda k.\mathbf{if}\ k < x \cdot n_t n_g + i + n_t j\ \mathbf{then}\ h(k)\ \mathbf{else}\ -, n_t n_b, i + n_t j)$$

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．

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 *reduceBlockTree* (以下単に *reduce* と書く) コードテンプレートを検証する．図 8 は *reduce* テンプレートの定義である [3]．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．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ている [4]．

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ため，ここでは各スレッド単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 5 (*reduce* テンプレート)**.**

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

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．

---

[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 節で議論する．

[4] 検証では，定義 3 に加えてユーザ関数の全域性，可換性及び結合性を仮定している．

本研究では、ループ中で配列アクセスパターンが複雑に変化するテンプレートの検証時に simulating function を用いた証明手法を提案する。一般にループのあるプログラムの検証にはループ不変式を与える必要がある。配列を操作するプログラムの場合、ループ不変式はループの各ステップの直後の配列の値を配列の初期値で表わしたような式である。一方、*reduce* テンプレートのようなプログラムは、ループ中でスレッドがアクセスする配列要素は複雑に変化する。そのようなプログラムのループ不変式を直接記述することは難しい。

提案する simulating function は，カーネルの実行を模倣するようなメタ論理の再帰関数である．一般に simulating function はプログラム中の while ループをプログラム状態に関する再帰関数として書き換えたような形で表現される．各ステップの直後の配列の値は simulating function を配列の添字に適用した値に一致するため，ループ不変式は容易に記述することができる．

reduce テンプレートの場合，simulating function は関数 $f_n(i)$ として記述できる．$f_n(i)$ 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り，以下のように定義される．

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

s_n は n ステップ目の s の値を表す．$f_{n+1}(i)$ は $reduceBody_{n+1}(2^{e_b-(n+1)})$ 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 6.

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

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．補題 6 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，$s_0 = 2^{e_b} \ge n_t \ge l$ 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 $f_{e_b}(0) = \sum_{i=0}^{l} f(i)$ が成り立つ．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．

$$
\begin{aligned}
\mathsf{BS}(i, 2n-2) &:= (\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto^1 f_n(i), \mathsf{array}_{1/n_t}(\mathbf{Sh}\ \mathtt{sdata}, n_t, f_n)) \\
\mathsf{BS}(i, 2n-1) &:= (\mathsf{array}_{1/n_t}(\mathbf{Sh}\ \mathtt{sdata}, n_t, -), \mathbf{Sh}\ \mathtt{sdata}[i] \mapsto^1 -)
\end{aligned}
$$

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

補題 7.

$$
\mathsf{BS}, i \vdash_T \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f_n(i) \wedge \mathtt{x1} = f_n(i)\} reduceBody_{n+1}(s_{n+1}) \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f_{n+1}(i) \wedge \mathtt{x1} = f_{n+1}(i)\}
$$

これは $f_{n+1}(i)$ と $reduceBody_{n+1}(s)$ の定義から容易に示せる．これを用いると定理 5 は以下のように示せる．

証明. *reduce* は $reduceBody_1(s_1); \ldots reduceBody_i(s_i); \ldots reduceBody_{e_b}(s_{e_b})$ と書けることから，補題 7 より

$$
\mathsf{BS}, i \vdash_T \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f_0(i) \wedge \mathtt{x1} = f_0(i)\} reduce \{\mathbf{Sh}\ \mathtt{sdata}[i] \mapsto f_{e_b}(i) \wedge \mathtt{x1} = f_{e_b}(i)\}
$$

f' として f_{e_b} をとれば，$f_{e_b}(0) = \sum_{i=0}^{l} f(i)$ より定理の結果を得る． □

```
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))
```

図 9: 最適化された *reduce* テンプレート

我々は *reduce* テンプレートの検証の過程で，図 9 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ことを発見した[5]．バリア同期の省略の他に，条件式に `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．このテンプレートに関する検証は省略するが，最適化前のテンプレートと同様に simulating function を用いることで検証と同様に行うことができている．

5 関連研究

GPUDSL は並列スケルトンを用いるプログラムの自動並列化コンパイラと考えることができるが，自動並列化コンパイラの検証として Bell によるもの [11] がある．健全な自動並列化の議論のために，彼らは Calculus of Communicating Systems 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．一方多くの GPUDSL のソース言語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 (型保存コンパイラ) が多く知られており，GPUDSL においても McDonell らによるもの [12] 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．

GPGPU カーネルの自動検証器は多く存在し，代表的なものとして Betts らによる検証器 GPUVerify [4] がある．GPUVerify はデータ競合や barrier divergence を検出する．本研究ではこれらの性質より強い計算の正しさを対象としている．

GPGPU カーネル検証のためのプログラム論理として，小島らによる SIMT (single instruction multiple threads) プログラムのための Hoare 論理 [13] がある．SIMT とは全スレッドが同期を取りながら一斉に実行をすすめる意味論である．GPUCSL はより一般的な全スレッドが任意の順序で実行するより一般的な意味論の上で健全性が証明されている．

6 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．我々は GPGPU カーネルを検証するためのライブラリ GPUVeLib を設計し，そのライブラリを用いてコードテンプレートを検証するためにコードテンプレートの正しさを定義し，この手法

[5] バリア同期の削除といった直感的に正当性を確認することが難しい最適化の正しさを形式検証によって確認できたことは本研究の貢献の 1 つである．この最適化の余地は検証の過程で発見したものであり，一般にはバリア仕様 BS や simulating function を形式的に記述することで不要なリソースの分配や不要な計算を検出し，冗長なバリア同期命令や無駄な計算を行うコードが発見できる可能性がある．

を Accelerate の map と reduce テンプレートに適用した．Reduce テンプレートの証明では simulating function を用いることで，GPGPU コードの正しさをメタ論理の関数の正しさに帰着できることを示し，テンプレートに最適化の余地があることを発見し最適化後のテンプレートの正しさを検証した．我々は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を考えているが，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 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 warp-synchronous programming，warp shuffle 命令といった機能が含まれる．

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なる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．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 [13] 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．

6.2 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 [10] や Cao ら [9] 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 [4] 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] NVIDIA. CUDA C Programming Guide, 2015.

[2] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, pp. 3–14, 2011.

[3] Bryan C Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In Proceedings of 16th PPOPP, pp. 47–56, 2011.

[4] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A Verifier for GPU Kernels. In Proceedings of OOPSLA, pp. 113–132, 2012.

[5] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical Symbolic Race Checking of GPU Programs. In Proceedings of SC, pp. 179–190, 2014.

[6] Stefan Blom, Marieke Huisman, and Matej Mihelčić. Specification and Verification of GPGPU Programs. Science of Computer Programming, Vol. 95, Part 3, pp. 376 – 388, 2014.

[7] Izumi Asakura, Hidehiko Masuhara, and Tomoyuki Aotani. Proof of Soundness of Concurrent Separation Logic for GPGPU in Coq. Journal of Information Processing, Vol. 24, No. 1, pp. 132–140, 2016.

[8] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In Proceedings of 32nd POPL, pp. 259–270, 2005.

[9] Jingyuan Cao, Ming Fu, and Xinyu Feng. Practical Tactics for Verifying C Programs in Coq. In Proceedings of the 2015 Conference on Certified Programs and Proofs, pp. 97–108, 2015.

[10] Adam Chlipala. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In Proceedings of 32nd PLDI, pp. 234–245, 2011.

[11] Christian J. Bell. Certifiably sound parallelizing transformations. In Certified Programs and Proofs, Vol. 8307 of Lecture Notes in Computer Science, pp. 227–242. 2013.

[12] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In Proceedings of the Symposium on Haskell, pp. 201–212, 2015.

[13] Kensuke Kojima and Atsushi Igarashi. A Hoare Logic for SIMT Programs. In Proceedings of 11th APLAS, Vol. 8301 of Lecture Notes in Computer Science, pp. 58–73, 2013.