平成30年度 修士論文

共有メモリ最適化のための GPGPU プログラム合成器

東京工業大学 情報理工学院 数理・計算科学系
学籍番号 17M30157
蟹 暁

指導教員
増原　英彦

平成31年2月1日

概要

本研究は GPGPU プログラムに対して，最適化されていないプログラムから，共有メモリを用いて最適化されたプログラムを半自動的に生成するプログラム合成器を提案する．

GPGPU プログラムの主な高速化の一つに共有メモリ最適化がある．共有メモリとはプログラマが明示的に利用する GPU 上の低容量なキャッシュメモリである．共有メモリ最適化とは次の手順を踏む最適化である．1) グローバルメモリ上の一部のデータを共有メモリにコピーする命令を挿入する．2) グローバルメモリを読み書きしていた計算をそのデータが共有メモリ上にあるならば共有メモリに対する読み書きに置き換える．

現実的なアプリケーションにおける共有メモリ最適化は，共有メモリにコピーするデータの範囲やコピーするタイミングに様々な可能性がある．また，共有メモリの適切な利用の仕方はプログラムを実際に実行しなければわからないことが多い．そのためプログラマは試行錯誤して最適化を行う必要がある．

一方で，共有メモリ最適化では複雑な条件式や添え字式を持つメモリアクセス式を用いることになる．そのため試行錯誤の際にプログラムの書き換えを手動で行う場合のプログラミングコストは高く，誤りのあるコードの原因になる．

最適化された GPGPU プログラムを得る方法として，逐次プログラムを入力して並列化されたプログラムを得る手法や，データ並列 DSL を用いる手法がある．しかし，これらはコンパイラが自動で最適化を行うものであり，ユーザは最適化の方針を指示することができない．

プログラマが最適化の方針を指示できる手法として，共有メモリ最適化をプログラム合成器を使って行う研究がある．しかし既存の合成器は汎用的な GPGPU プログラムを対象としていたため，小さな数値計算プログラムの最適化でも数分以上の時間を要していた．

そこで我々は共有メモリ最適化に特化した合成器 PSySha を提案する．PSySha は最適化前のプログラムのプロファイル情報を収集し，共有メモリアクセス式が満たすべき制約を生成することで，より小さな合成問題に帰着させる．それを独自に作成したソルバに解かせることで合成を完了する．PSySha は共有メモリに袖領域をコピーする場合，コピーしない場合

やテンポラルブロッキング最適化のような、異なる複雑な最適化に対して適用が可能である．

PSySha を用いて 3 次元熱拡散方程式や姫野ベンチマークに対して上記の最適化を対象に合成を行なった．熱拡散方程式では従来の合成器よりも 10 倍以上の合成の高速化を達成した．また，従来の合成器では合成ができなかった姫野ベンチマークのような現実的なプログラムに対しても，数秒程度で合成できることを確認した．

謝辞

本修士論文は以下の方々なくして，存在しえなかったでしょう．

本研究を進めるにあたり，本論文を何度もご精読頂き多大なる有用なコメントを頂きました増原英彦教授，数多くのアドバイスをくださった青谷知幸助教に深く感謝いたします．

最後となりますが発表練習に出席していただき，多くの助言をくださった研究室の皆様に心より感謝いたします．

目 次

第 1 章 はじめに **1**
1.1 共有メモリ最適化 1
1.2 プログラム合成を用いる根拠 2
1.3 本論文の概要 3

第 2 章 背景 **5**
2.1 共有メモリを用いた GPGPU プログラムの最適化 5
2.2 GPGPU プログラム合成器 7

第 3 章 問題 **9**
3.1 要約 9
3.2 最適化方針の指示 9
3.3 最適化のための計算時間 10

第 4 章 提案: 共有メモリ最適化のためのプログラム合成器 **13**
4.1 利用手順 13
4.2 提案手法の実現 15
4.2.1 プロファイル情報を記録する合成手法を採用する根拠 15
4.2.2 記録するプロファイル情報 16
4.2.3 候補式集合の生成 18
4.2.4 探索プログラム 18
4.2.5 合成されるプログラムの正しさ 19
4.2.6 競合状態になるプログラムの合成 20
4.3 様々な共有メモリ最適化への対応 21
4.3.1 例 1: 袖領域をコピーするような共有メモリ最適化 . 21
4.3.2 例 2: テンポラルブロッキング最適化 24

第 5 章 実現 **27**

第 6 章 実験 **29**
6.1 既存の合成器との合成時間の比較 29
6.2 入力データの大きさに対する合成時間 30
6.3 記録する変数の数に対する合成時間 32

6.4 合成されたプログラムの性能 33

第 7 章 関連研究 **37**

7.1 GPUDSL 37

7.2 逐次プログラムの自動並列化 37

7.3 GPGPU プログラム合成器 38

第 8 章 結論と課題 **39**

8.1 結論 39

8.2 今後の課題 39

付 録 A 実験で用いたプログラム **45**

図 目 次

2.1 2 次元 5 点ステンシルカーネル，入力データの各要素をその近傍要素を元に更新する． 5
2.2 共有メモリ最適化のパターン 7

3.1 合成したい式 10
3.2 図 3.1 に対応する下描き部分の一例 10

4.1 PSySha の合成アルゴリズム 16

6.1 入力データの大きさに対する姫野ベンチマークの合成時間 (sec) 31
6.2 記録する変数の数に対する姫野ベンチマークの合成時間 (sec) 32
6.3 PSySha で合成した場合と CUDA で直接記述した場合の姫野ベンチマークプログラムの処理性能 (GFLOPS) 34
6.4 CUDA で直接記述した場合と PSySha で合成した場合の斜め方向の袖領域のコピーの仕方 34

表 目 次

4.1 ソースコード 4.2 を入力が長さ 64 の配列，1 ブロックが 4×4 の 16 スレッド，2 × 2 の 4 ブロックで実行した場合のプロファイル情報 17

6.1 合成器の実行環境 30
6.2 GPGPU プログラムの実行環境 30
6.3 Kani-CUDA と PSySha の合成時間 (sec) の比較 31

第1章　はじめに

本研究はGPGPUプログラムに対して，最適化されていないプログラムから，共有メモリを用いて最適化されたプログラムを半自動的に生成するプログラム合成器を提案する．

GPGPU (General-Purpose computing on Graphics Processing Unit) はGPU (Graphics Processing Units) と呼ばれる画像処理向けの演算装置を，汎用計算に応用することである．並列性の高い問題に対しては，CPUよりも電力効率が良い高性能計算を可能とするため，様々な用途に用いられている．

GPGPUを利用するための代表的な言語として，C言語の拡張であるCUDA Cがある．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コードからなる． GPUはプログラマが指定した構造のスレッドグリッドを作成し，カーネルコードを各スレッドで実行する．グリッドは複数のブロックで構成され，ブロックは複数のスレッドで構成される．

高性能なGPGPUプログラムを記述するにはGPUプログラミング特有の最適化が求められる．例えば，共有メモリやテクスチャメモリなどのキャッシュメモリの利用 [11, 7] やワープ発散[1]の回避 [12]，コアレスアクセス[2]を発生させる [5] などである．

1.1　共有メモリ最適化

我々の研究は共有メモリを用いた最適化に注目する．共有メモリとはブロック単位でしか共有されないGPU上のキャッシュメモリであり，低容量だがレイテンシはグローバルメモリよりも100倍程度小さい．そのため複数のスレッドが同一のグローバルメモリにあるデータを読み書きするようなプログラムは，一時的にそのデータを共有メモリに格納してから読み書きを行うことで，プログラムの高速化が期待できる．共有メモリはCPU

[1]ワープ発散とはワープ内のスレッドが異なる実行パスをとることである．ワープとは同じ命令を同時に実行されるスレッドグループであり，CUDAはスレッドブロックをワープに分割した上で実行する．

[2]コアレスアクセスとはワープ内のスレッドが全て隣あったメモリアドレスにアクセスすることである．このアクセスパターンはメモリレイテンシが小さい．

上のキャッシュメモリのように自動的に読み書きを行うものではなく，プログラマが明示的に読み書きをするものであることに注意されたい．

本研究において共有メモリ最適化とは以下の手順を踏む最適化である．1) グローバルメモリ上の一部のデータを，各ブロックの共有メモリに並列にコピーする命令を挿入する．2) グローバルメモリを読み書きしていた計算を，そのデータが共有メモリ上にあるならば共有メモリに対して読み書きする計算に置き換える．

しかし，共有メモリは各ブロックに個別に共有される小容量なメモリであるため，これを用いて最適化した GPGPU プログラムは複雑な記述を伴うことが多い．具体的にはメモリアクセス式に，複雑な条件分岐や添え字式を持つメモリアクセス式が追加されることになる．

現実的なアプリケーションにおける共有メモリ最適化は，共有メモリにコピーするデータの範囲やコピーを行うタイミングに様々な可能性があるため，プログラマは試行錯誤して最適化を行う必要がある．例えば，丸山らは単純なステンシル計算プログラムに対して様々な最適化を施し，Fermi M2075 と Kepler K20X の二つの GPU 上での性能を計測した [7]．そのうち共有メモリを用いた最適化は 4 パターンあった．その一つである時間ブロッキング最適化はどちらの環境でも高い性能が出たが，その他の最適化は Fermi では有益ではなかった．一方で Holewinski らは時間ブロッキング最適化は 3 次元 Jacobi カーネルであまり性能が出ないという報告をしている [4]．

一方，前述のように，共有メモリを用いた最適化がなされたプログラムはしばしば記述が複雑になる．また，共有メモリの利用の仕方によって，プログラムの書き換えを手動で書き換える場合のプログラミングコストは高く，誤りのあるコードの原因になる．

そこで我々は最適化の方針はプログラマが決め，複雑なプログラムの詳細を自動で生成するような半自動的な手法に注目した．最適化の方針をプログラマに任せる根拠は，共有メモリの適切な利用の仕方がプログラムを実際に実行しなければわからないことが多いからである．我々は共有メモリを用いていない CUDA プログラムに対して，グローバルメモリから共有メモリへのコピーする式を記述することで，グローバルメモリアクセス式を共有メモリアクセス式に自動で置き換えるアプローチをとる．

1.2　プログラム合成を用いる根拠

最適化を行うアプローチは大きく次の三つに分類される．1) プログラマが CUDA[10] や OpenCL[9] などの低級な言語で直接記述する．2)Accelerate[2] や Ikra[8] などのデータ並列 DSL によって最適化されたコードを出力させ

る．3) 最適化前のプログラムを変換して最適化されたプログラムを得る．3) のプログラムを変換するアプローチはさらに，逐次プログラムの自動並列化 [15, 14] やプログラム合成 [1] などに分けられる．

我々の研究はプログラム合成に注目する．プログラム合成とは，プログラマが合成器にプログラムが満たすべき仕様を与えることで仕様を満たす実行可能なプログラムを自動生成する技術のことである．プログラム合成の一種に syntax-guided synthesis (SyGuS)[1] がある． SyGuS は，満たすべき仕様に加えて，探すべきプログラムの空間を規定する文法を与えて合成する手法で，より現実的な大きさのプログラムの合成が可能である．

データ並列 DSL や逐次プログラムの自動並列化において，共有メモリを用いた最適化をするのはコンパイラの役割であり，それらのユーザは最適化に関与しない．我々が研究対象とするのはユーザが最適化の方針を決め，その詳細を自動で記述するような半自動的な最適化手法である．これは SyGuS のように，プログラムの概要から完全なプログラムを生成するという性質と相性が良いと考えられる．

1.3　本論文の概要

本論文の以降は，次のような構成である．第 2 章は研究背景，第 3 章では GPGPU プログラミングを援助する既存研究の問題点，第 4 章では提案する合成器と合成アルゴリズムを説明する．第 5 章では合成器の実現，第 6 章では合成器を用いて実際にプログラム合成を行った実験，第 7 章では関連研究， 第 8 章では結論と今後の課題について述べる．

第2章　背景

2.1　共有メモリを用いたGPGPUプログラムの最適化

GPGPU プログラムの最適化の一つに共有メモリを用いたものがある．この最適化は，1 ブロック内で同じグローバルメモリアドレスに複数回アクセスするような式を，共有メモリアクセス式に置き換えることによって高速化する．

しかし，この最適化を行うにはメモリアクセス式を複雑な条件式や添え字式を使って書き換える必要がある．図 2.1 とソースコード 2.1 に示す単純な 2 次元 5 点ステンシル計算プログラムを例に説明する．ステンシル計算とは入力データの各要素を，その要素の近傍要素を用いて計算した値で更新する処理である．このプログラムは更新値を求める要素ごとにスレッドを対応させている．

ソースコード 2.2 は共有メモリを用いて最適化されたステンシルカーネルである．この最適化は入力データを複数のブロックに分割し，それぞれの要素を共有メモリにコピーし，共有メモリから各要素を読み込み更新値を計算する．9 行目から 11 行目のメモリアクセス式は近傍要素を取得している部分だが，その要素がブロックや入力データの境界にあるかどうかによって，異なる配列の要素を複雑な添え字式で参照しなければならない．

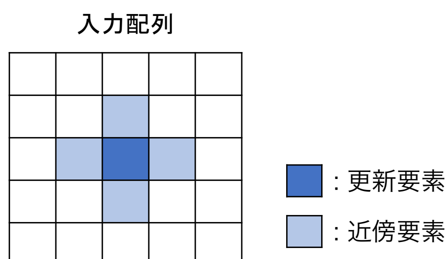

図 2.1: 2 次元 5 点ステンシルカーネル，入力データの各要素をその近傍要素を元に更新する．

ソースコード 2.1: 最適化前の 2 次元 5 点ステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      out[c] = in[(i == 0) ? c : c - 1]
6          + in[(i == nx - 1) ? c : c + 1]
7          + in[(j == 0) ? c : c - nx]
8          + in[(j == ny - 1) ? c : c + nx]
9          + in[c];}
```

ソースコード 2.2: 2.1 を共有メモリ最適化したプログラム

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      int csb = thradIdx.x + threadIdx.y *blockDim.y;
6      __shared__ float sb[blockDim.x*blockDim.y];
7      sb[csb] = in[c];
8      __syncthreads();
9      out[c] = ((threadIdx.x != 0 || i == 0)
10             ?  sb[(i == 0) ?  csb :  csb - 1]
11             :  in[c - 1])
12         + ((threadIdx.x != blockDim.x - 1 || i == nx - 1)
13             ?  sb[(i == nx - 1) ?  csb :  csb + 1]
14             :  in[c + 1] )
15         + ((threadIdx.y != 0 || j == 0)
16             ?  sb[(j == 0) ?  csb :  csb - blockDim.x]
17             :  in[c - nx])
18         + ((threadIdx.y != blockDim.y - 1 || j == ny - 1)
19             ?  sb[(threadIdx.y==blockDim.y-1) ?  csb :  csb+blockDim.x]
20             :  in[c + nx])
21        + sb[csb];}
```

また，この共有メモリを用いた最適化は一意ではない．スレッドブロックの宣言の仕方や，共有メモリの利用の仕方のパターンの一例を図 2.2 に示す．図 2.2 はどのスレッドがどの要素をコピーするかという関係を表している．A) はソースコード 2.2 のように更新要素に対応する入力要素だけを共有メモリにコピーしているが，B) は更新する領域よりひとまわり大きな領域の要素までを共有メモリにコピーする．この場合，A) と比較して共有メモリを読む式は単純な記述になるが，共有メモリにコピーする式が複雑になる．C) は A) や B) よりもひとまわり大きなスレッドブロックを宣言し，B) と同様に更新領域よりひとまわり大きな領域を共有メモリにコピーする．C) は共有メモリへのコピーする式は単純になるが，共有メモリから読む際に，動作するスレッドを限定するために条件分岐を使う必要がある．

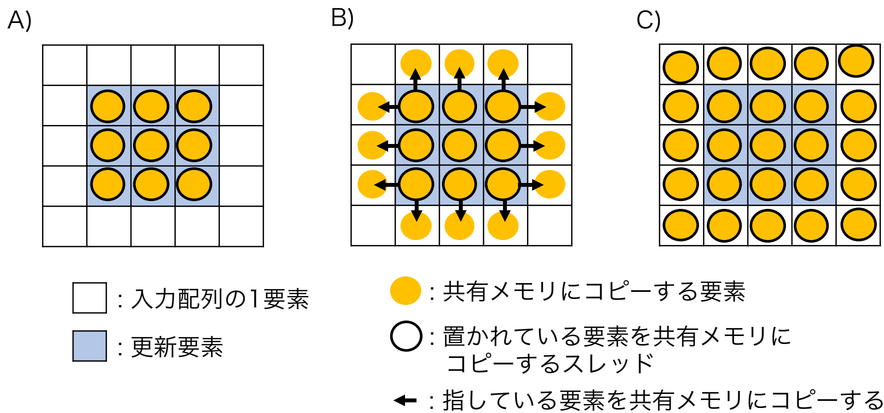

図 2.2: 共有メモリ最適化のパターン

2.2 GPGPU プログラム合成器

GPGPU を対象とするプログラム合成を用いた支援に， OpenCL を対象とした SynthCL[13] や CUDA を対象とした Kani-CUDA[16] がある．これらはプログラム合成器であり，逐次プログラムや最適化されていないプログラムを仕様として与えると，それと同じ計算結果を返すプログラムを合成することができる．

これらの合成器を利用する際，まず，ユーザは「下描き」と呼ばれるプログラムの概形を記述する．ユーザは，最適化されたプログラムのうち，記述が面倒な一部分だけを「穴」にして下描きを記述できる．次にユーザはプログラムが満たすべき仕様を合成器に入力する．この仕様とは，例えばプログラムの入出力の例などである．合成器は与えられた制約をみたすように，その穴を正しく埋めたプログラムを合成することができる．

SynthCL と Kani-CUDA はともに Rosette[13] で実装されている．Rosette とはプログラム合成やプログラム検証などの機能を持った領域特化言語 (DSL) を容易に開発することを目的としたプログラミング言語で，合成したい言語のインタプリタを記述するだけでその言語の合成器を作成することができる．Rosette は実装の詳細を抽象化するための記号的な値を持ち，プログラムの記号実行が可能である．また Rosette は成り立って欲しい性質を記述するための制約文を持つ．この制約文と下描きの記号実行のトレースから，課せられる制約を生成し，制約を SMT ソルバに解かせることによって合成を実現している．

第3章　問題

3.1　要約

共有メモリ最適化のための既存の手法には以下の 3 つの問題点がある．

- 自動並列化やデータ並列 DSL において，共有メモリを用いて最適化されたプログラムを記述するのはコンパイラの役割であるため，プログラマが最適化の方針を指示することができない．
- プログラマがより詳細まで最適化の方針を指示できる GPGPU プログラム合成器があるが，それらは汎用的な GPGPU プログラムを対象としているため，現実的な数値計算プログラムが合成できない．また，小さな数値計算プログラムであっても 5 分以上の合成時間を要することがある．
- GPGPU プログラム合成器において，下描きに与える文法を工夫することで現実的な時間で最適化プログラムを合成させることは不可能ではないが，そのような文法を記述する手間は，最適化されたプログラムを記述することに匹敵するほど大きくなりかねない．

以下にそれぞれの問題点を詳しく議論する．

3.2　最適化方針の指示

最適化された GPGPU プログラムを容易に得る方法として，逐次プログラムの自動並列化やデータ並列 DSL の研究がされている [2, 8, 14]．しかし，これらの研究では共有メモリを用いて最適化されたプログラムを記述するのはコンパイラの役割であるため，プログラマが最適化の方針を指示することができない．

これらの研究においてプログラマがどのような指示を出すことができれば良いかを述べる．共有メモリ最適化は大きく二つの手順に分けられる．1) グローバルメモリ上のデータを共有メモリにコピーする式を記述する手順と 2) 共有メモリを読み込む式を記述する手順である．どちらか一方の方針が定まれば，他方の方針はある程度定まるはずである．ここで「あ

る程度」と述べているのは，その方針を実現するプログラムが複数存在するからである．例えば，共有メモリを読み込む式が具体的に定まれば，共有メモリにはグローバルメモリのどの部分をコピーすれば良いかは定まるが，コピーするタイミングやどのスレッドがコピーを担当するかなどに選択肢が存在する．著者は，様々な共有メモリ最適化を試す際，プログラマは 1) と 2) のどちらか一方の手順について方針を指示できれば，後のプログラムは自動的に埋められるのではないかと考えた．方針を指定するとは，例えば，ソースコード 2.2 を例にあげると，7 行目の共有メモリへのコピーする段階でどのスレッドでどの部分をコピーするかということや，10 から 13 行目のメモリアクセス式の条件分岐の有無や添え字式などをユーザが指定できれば良い．

3.3 最適化のための計算時間

より低レベルな GPGPU プログラミングを援助する研究としてプログラム合成 (SyGuS) を用いた研究がされている [13, 16]．しかし，これらは汎用的な GPGPU プログラムを対象としているため，現実的な数値計算プログラムが合成できない．また，合成対象が小さな数値計算プログラムであっても合成時間が現実的でない．

既存の GPGPU プログラム合成器である Kani-CUDA[16] を用いてソースコード 2.2 のように最適化されたステンシル計算プログラムを合成する場合を例に説明する．Kani-CUDA のユーザは「穴」と呼ばれるその部分を埋める候補式の列挙を用いて下描きを記述する．例えば，プログラム中のメモリアクセス式を合成器を用いて図 3.1 のような式に書き換えたい場合，ユーザは図 3.2 のような式を下描きに記述する．?で括られた部分が穴を表す．図 3.2 を「穴を持つメモリアクセス式」と呼ぶ．つまりユーザが記述する下描きはソースコード 3.1 のようになる．これに加えて，ユーザはある大きさの任意の入力データに対し，最適化前のプログラムと出力が同じであるという仕様を与える．下描きと仕様から最適化されたプログラムが合成される．入力データの大きさが 6×6 を指定して合成したところ，全ての穴を埋めるのに約 5 分要した．

```
(threadIdx.x == 0 && i != 0)
  ? in[c-1]
  : sb[(i == 0)
        ? csb
        : csb - 1]
```

図 3.1: 合成したい式

```
(threadIdx.x == (? 0 (blockDim.x - 1) (? && ||) i == (? 0 nx))
  ? in[c-1]
  : sb[((? threadIdx.x i) == (? 0 ((? nx blockDim.x) - 1))
        ? csb
        : csb - (? 1 blockDim.x)]
```

図 3.2: 図 3.1 に対応する下描き部分の一例

ソースコード 3.1: ステンシルカーネルの下描き

```
1  __global__ void stencil_kernel_sketch(
2      float* in, float* out,
3      int nx, int ny,) {
4    int i = blockDim.x * blockIdx.x + threadIdx.x;
5    int j = blockDim.y * blockIdx.y + threadIdx.y;
6    int c = i + j * nx;
7    __shared__ float sb[blockDim.x*blockDim.y];
8    int scb = threadIdx.x + threadIdx.y * blockDim.x;
9    sb[scb] = in[c];
10   syncthreads();
11   out[c] = sb[scb]
12     + [穴を持つメモリアクセス式]
13     + [穴を持つメモリアクセス式]
14     + [穴を持つメモリアクセス式]
15     + [穴を持つメモリアクセス式];
16 }
```

下描きに与える文法を工夫することで現実的な時間で最適化プログラムを合成させることは不可能ではないが，そのような文法を記述する手間は，最適化されたプログラムを記述することに匹敵するほど大きくなりかねない．

まず文法の工夫の仕方について説明する．文法の工夫の仕方として著者は分割合成 [17] を提案している．分割合成とは，互いに依存関係のないグローバルメモリアクセス群を一つずつ順に合成する手法である．互いに依存関係のないとは，最適化前のプログラム片と最適化後のプログラム片がそれぞれ個別に置き換えても処理の結果が変わらないことを言う．穴を持つ各メモリアクセス群を制御変数によって最適化前のメモリアクセスに切り替えるコードで囲むことにより，分割合成を実現した．

例えば，ソースコード 3.1 に分割合成を適用しようとした場合の下描きはソースコード 3.2 のようになる．`switch` が制御変数で，合成が 1 反復完了するごとに `switch` の値を切り替えて，計 4 回の合成をして，全体の合成を完了する．この分割合成を適用した場合 10 秒程度で合成が完了できる．しかし，実際に利用するようなプログラムに対して分割合成を適用しようとすると，下描きの記述は煩雑になり，コードの誤りを招く原因となる．

ソースコード 3.2: 分割合成する場合のステンシルカーネルの下描き

```
1  __global__ void stencil_kernel_sketch(
2      float* in, float* out,
3      int nx, int ny,) {
4    int i = blockDim.x * blockIdx.x + threadIdx.x;
5    int j = blockDim.y * blockIdx.y + threadIdx.y;
6    int c = i + j * nx;
7    __shared__ float sb[blockDim.x*blockDim.y];
8    int scb = threadIdx.x + threadIdx.y * blockDim.x;
9    sb[scb] = in[c];
10   syncthreads();
11   out[c] = sb[scb]
12     + ((switch == 0)
13         ? 穴を持つメモリアクセス式
14         : in[(i == 0) ? c : c - 1] )
15     + ((switch == 1)
16         ? 穴を持つメモリアクセス式
17         : in[(i == nx - 1) ? c : c + 1])
18     + ((switch == 2)
19         ? 穴を持つメモリアクセス式
20         : in[(j == 0) ? c : c - nx])
21     + ((switch == 3)
22         ? 穴を持つメモリアクセス式
23         : in[(j == ny - 1) ? c : c + nx]);
24 }
```

第4章　提案: 共有メモリ最適化のためのプログラム合成器

我々は共有メモリを用いた最適化に特化したGPGPUプログラム合成器PSySha (Profile-based Synthesizer for Shared memory opitimization)を提案する．PSyShaのユーザは，グローバルメモリ上の配列を用いて素朴に記述されたCUDAプログラムに，最適化の方針を書き加える．そうするとPSyShaは自動的に，グローバルメモリからの読み込みを共有メモリからの読み込みに置き換えたプログラムを合成する．最適化の方針は，共有メモリ宣言とグローバルメモリから共有メモリへのコピー式を書くことで指示する[1]．本章ではPSyShaを用いたプログラムの最適化手法やその原理について説明する．

4.1　利用手順

PSyShaを用いてGPGPUプログラムを最適化する手順の一つを2次元5点ステンシル計算を例に説明する． GPUスレッドによって実行されるカーネル関数`stencil_kernel`は，グローバルメモリ上の配列`in`を引数として受け取り，`in`上の各要素についてのステンシル計算を行う．

ソースコード4.2の赤字部分がユーザの記述する部分であり，ユーザは5行目から7行目のようなグローバルメモリから共有メモリへコピーする式と，最適化したいメモリアクセス式に8行目のように`__opt__`という注釈を挿入する．5行目が共有メモリ上に配列`sb`を確保する宣言で，6行目は各スレッドに，自身が担当する`in`の要素を`sb`にコピーさせている．この段階のプログラムのことを以下では「注釈付きプログラム」と呼ぶこととする．注釈付きプログラムをPSyShaに入力すると，PSyShaは共有メモリからデータを読み込むようなメモリアクセス式を自動生成する．ソースコード4.3の赤字部分である8行目から12行目が生成されたメモリアクセス式である．

この例では，実行時の条件によって，元々のグローバルメモリアクセス式と，共有メモリアクセス式を使い分ける式が生成されている．これはア

[1]コピー式を書く代わりにグローバルメモリを参照する式を共有メモリを参照する式に置き換える方法も可能である．これについては4.3.1節で説明する．

クセスされるグローバルメモリアクセスの全てが共有メモリにコピーされていないためである．このような条件式も自動的に生成していることに注意されたい．

ソースコード 4.1: 最適化前のステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      out[c] = in[(i == 0) ? c : c - 1]
6          + in[(i == nx - 1) ? c : c + 1]
7          + in[(j == 0) ? c : c - nx]
8          + in[(j == ny - 1) ? c : c + nx]
9          + in[c];}
```

ソースコード 4.2: 注釈付きステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      __shared__ float sb[blockDim.x *blockDim.y];
6      int csb = thradIdx.x + threadIdx.y *blockDim.y;
7      sb[csb] = in[c];
8      __syncthreads();
9      out[c] = __opt__.in[(i == 0) ? c : c - 1]
10         + __opt__.in[(i == nx - 1) ? c : c + 1]
11         + __opt__.in[(j == 0) ? c : c - nx]
12         + __opt__.in[(j == ny - 1) ? c : c + nx]
13         + __opt__.in[c];}
```

ソースコード 4.3: PSySha によって生成されるステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      __shared__ float sb[blockDim.x*blockDim.y];
6      sb[thradIdx.x + threadIdx.y * blockDim.y] = in[c];
7      __syncthreads();
8          out[c] = ((threadIdx.x != 0 || i == 0)
9              ?  sb[(i == 0) ?  csb :  csb - 1]
10             :  in[c - 1])
11         + ((threadIdx.x != blockDim.x - 1 || i == nx - 1)
12             ?  sb[(i == nx - 1) ?  csb :  csb + 1]
13             :  in[c + 1] )
14         + ((threadIdx.y != 0 || j == 0)
15             ?  sb[(j == 0) ?  csb :  csb - blockDim.x]
16             :  in[c - nx])
17         + ((threadIdx.y != blockDim.y - 1 || j == ny - 1)
18             ?  sb[(threadIdx.y==blockDim.y-1) ?  csb :  csb+blockDim.x]
19             :  in[c + nx])
```

```
20        + sb[csb];}
21 }
```

4.2 提案手法の実現

PSySha におけるメモリアクセス式の合成のアルゴリズムを説明する．PSySha は最適化前のプログラムのプロファイル情報からメモリアクセス式レベルの小さな合成問題に帰着させる．そして，合成問題を独自に開発したソルバによって解き，合成を実現している．

PSySha は`__opt__`がついたメモリアクセス群は分割合成 [17] を用いて一つずつ順に合成する．分割合成とは，複数のメモリアクセスをそれぞれ個別に合成する手法であり，互いに依存関係のないメモリアクセス群を最適化する際に有効である．互いに依存関係がないとは，それぞれのメモリアクセス式を最適化前後で個別に置き換えても，プログラムの出力結果が変わらないことを表す．

PSySha はまず最適化前のプログラムを一度だけ実行して，`__opt__`がついたメモリアクセス式ごとにプロファイル情報を記録し，メモリアクセス式ごとの合成問題をそれぞれ生成する．その後，ソルバはそれぞれの合成問題を逐次的に処理する．以下では，一つのメモリアクセス式の合成アルゴリズムを説明する．

図 4.1 は PSySha が入力である最適化前のプログラムを受け取ってから合成を完了するまでの処理の概要を表している．この処理は大きく分けて三つに分別される．(1)PSySha はまず入力された最適化前のプログラムから，メモリアクセス式の場所で参照できる変数名を抽出する．さらに，小さなデータを使ってプログラムを Kani-CUDA で実行し，メモリアクセスがあったときの各変数の値などを記録する．(2) その後，記録した変数などから候補式となる条件式や添え字式の集合を生成する．(3) 最後に，生成した候補式の集合から仕様を満たす式を探索する．ここでいう仕様とは，最適化前のメモリアクセス式と同じ結果を与えるということである．以下ではプロファイル情報を利用するようなアルゴリズムを選択した根拠とそれぞれの段階の詳細について説明する．

4.2.1 プロファイル情報を記録する合成手法を採用する根拠

プロファイル情報を記録する合成手法を採用する理由は，合成問題をより小さな合成問題に帰着させるためである．

既存の GPGPU プログラム合成器で最適化する場合，入力データに記号値を用いて，ある大きさの任意の入力に対して最適化前と最適化後のプ

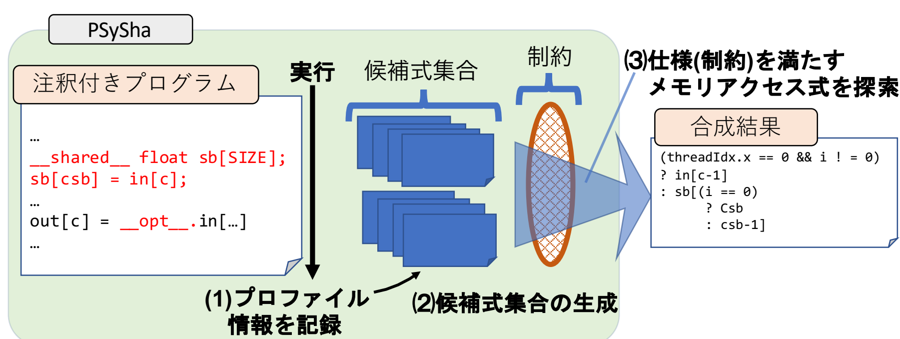

図 4.1: PSySha の合成アルゴリズム

ログラムの出力が等しいという仕様を与えていた．合成器は下描きの記号実行トレースと仕様から制約を生成し，制約をソルバに解かせることで合成を実現させていた．しかし，プログラム全体の入出力が等しいという仕様を与えていたため，現実的な大きさのプログラムに対しては生成される制約が非常に大きいものとなり，合成時間が数分程度，もしくは合成が完了できない例もあった．

共有メモリ最適化は主にメモリアクセス式の置き換えであり，添え字式や条件式が適切な値をとるかどうかという，より小さな問題に帰着できる．

そこで我々は最適化前のプログラムを実行し，そのプロファイル情報から新たに添え字式レベルの小さな合成問題を定式化するアプローチを選択した．

4.2.2　記録するプロファイル情報

PSySha は候補式集合や探索時に用いる制約を生成するために，プロファイル情報を記録する．プロファイル実行の起動と，実行の記録は次のようにして行われる．

まずユーザはメモリアクセス式に注釈を付けた CUDA カーネルをプロファイル実行用データを作成して起動するホストコードを記述する．プロファイル実行用データとしては，記号値[2]で埋められた小さな配列を作ることを想定している．また，CUDA カーネルも最小限のスレッド数で起動することを想定している．これらの配列の大きさやスレッド数については 4.2.5 節で議論する．

各スレッドで注釈の付いたメモリアクセス式を評価する際に，以下の値を記録する．

[2]PSySha は Rosette 上に作成されており，Rosette の記号値を生成することができる．

1. スレッド ID
2. ブロックの次元
3. ブロック ID
4. グリッドの次元
5. 参照できる変数のそれぞれの値
6. 共有メモリ上の配列の「正解」となる添え字番号

6 について補足する．これは注釈のついたグローバルメモリアクセス式を実行した結果得られる値を X，共有メモリ上の配列を S とした時 S[i]=X となるような i を探して得る[3]．

したがって，ソースコード 4.2 を入力が 8×8 の二次元空間を表す長さ 64 の配列，1 ブロックが 4×4 の 16 スレッド，2×2 の 4 ブロックで実行した場合は表 4.1 のような情報が記録される．ここでは簡単のため，`threadIdx.x` や `blockDim.x` は `tidx`，`bdimx` と略して記す．

`tidx`	`tidy`	`bidx`	`bidy`	`bdimx`	`bdimy`	`i`	`j`	`c`	正解の添え字
0	0	0	0	4	4	0	0	0	0
1	0	0	0	4	4	1	0	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	0	0	0	4	4	3	0	3	NF
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	3	1	1	4	4	3	3	63	2

表 4.1: ソースコード 4.2 を入力が長さ 64 の配列，1 ブロックが 4×4 の 16 スレッド，2×2 の 4 ブロックで実行した場合のプロファイル情報

NF という表記は正解の添え字番号が存在しないことを表す．以降では 1 行分のデータを「事例」と呼ぶ．

2，4，6 の値は記録をとるかどうかをユーザが制限できる．プログラム中の参照可能な変数や定数を全て記録した場合，生成される候補式空間が大きくなり，合成時間が長くなる．これを防ぐためにユーザは記録を取る変数を制限する必要がある．ユーザが記録をとる変数を選ぶ際の留意点の一例を述べる．例えば，ステンシル計算の最適化に用いる場合には，入力データの次元や共有メモリの次元などを基準に変数や定数を選ぶことが有

[3] そのような i は複数ある場合もあり得るが，現在は最初に見つかったものを正解としている．

効だと考えられる．なぜなら，スレッドが注目している共有メモリ上の要素とその近傍要素の添字番号の差分は共有メモリの次元の倍数を含んだ式が用いられることが多く，また，それらがブロックの境界条件にも用いられることも多いからである．

4.2.3　候補式集合の生成

PSySha は記録した情報から候補式集合を生成する．合成したいメモリアクセス式は以下のテンプレートに当てはめる．

$$\text{(グローバルメモリアクセス式が共有メモリアクセス式に置き換え可能な条件)} \tag{4.1}$$
$$\texttt{?}\ \texttt{sb[}\text{(共有メモリ添え字式)}\texttt{]} \tag{4.2}$$
$$\texttt{:}\ \text{(元のグローバルメモリアクセス式)} \tag{4.3}$$

ここで合成すべきは，(4.1) 式の論理式と (4.2) 式の添え字式である．したがって，`int` 型である算術式と `boolean` 型である論理式の候補式集合を生成する必要がある．二つの型の候補式集合をそれぞれ生成する．

算術式集合は以下の文法規則で生成される．

$$\begin{array}{rcl} s \in \texttt{Sym} & & \text{(実行時に記録する変数やスレッド }\texttt{ID}\text{ などの定数)} \\ p \in \texttt{Prm} & ::= & 0 \mid 1 \mid 2 \mid s \mid 2 * s \mid s * s \mid 2 * s * s \\ \mathit{exp} \in \texttt{Exp} & ::= & \mathit{exp}\ \texttt{+}\ p \mid \mathit{exp}\ \texttt{-}\ p \end{array}$$

s は前節で挙げた変数らを表す．生成される式の次元は 2 以下で，利用可能な定数は 0，1，2 に限定する．また exp は最大項数を 5 に制限して生成している．

論理式集合は以下の文法規則で生成される．

$$\begin{array}{rcl} b \in \texttt{BPrm} & ::= & \mathit{exp}\ \texttt{==}\ \mathit{exp} \mid \mathit{exp}\ \texttt{!=}\ \mathit{exp} \mid \mathit{exp}\ \texttt{<}\ \mathit{exp} \mid \mathit{exp}\ \texttt{<=}\ \mathit{exp} \\ \mathit{bexp} \in \texttt{BExp} & ::= & b\ \texttt{||}\ b \mid b\ \texttt{\&\&}\ b \mid b \end{array}$$

生成規則の exp は先述の算術式の生成規則によって生成される項数 2 以下の算術式を表す．

4.2.4　探索プログラム

PSySha は前節で生成した候補式集合から仕様を満たす式を探索する．候補式集合の各候補式について，全事例で適切であるかどうかということ

をテストして探索する．ある事例で不適切であった時点でその式のテストを打ち切る．

まずは以下の論理式が全スレッドで真となるような条件式 $bexp \in$ BExp を探索する．ここではある事例で $bexp$ が適切であるとは以下を満たすことを言う．smid は共有メモリの「正解」の添え字番号を表す．

$$bexp \texttt{ == } (\texttt{smid != } \mathrm{NF}) \tag{4.4}$$

$bexp$ は「正解」となる添え字番号が存在する事例で真となればよい．つまり，smid が NF でない時に $bexp$ は真を取ればいい．

続いて，共有メモリアクセス式の添え字式 $exp \in$ Exp の探索について説明する．exp がある事例で適切であるとは以下を満たすことを言う．

$$(\texttt{smid != } \mathrm{NF}) \texttt{ || } (exp \texttt{ == smid}) \tag{4.5}$$

合成される式は，正解の添字番号 smid が存在するならば，smid と同じ値を取れば良い．したがって満たすべき条件は上記のようになる．ここで 4.2 式の添え字式が次のような形をしていた場合の探索方法を説明する．

$$4.2 \text{ 式の添え字式} \equiv cond \texttt{ ? } then \texttt{ : } else \tag{4.6}$$

この場合，先述した単純な探索方法だと三項演算子を用いた算術式の探索空間の集合は生成していないため，このような式は合成できない．単純な探索方法で見つからなかった場合は次の探索手順で (4.6) 式のような形の式を合成する．

まずは記録したプロファイル情報の事例数に対して，不適切である事例数の限界値 limit を与える．その上でまず $then$ 式を最初に探索する．候補式のテスト中にその式が不適切である事例が limit 数見つかった時点でその式のテストを打ち切る．不適切である事例数が limit を超えなければ，その式を $then$ 式として決定する．次に $else$ 式の探索に移る．$else$ 式の探索は $then$ 式が不適切であった全ての事例において適切であるような式を探索する．最後に $cond$ 式を探索する．$cond$ 式は $then$ 式が適切であった事例で真，そうでない事例では偽となるような式を探索する．

4.2.5 合成されるプログラムの正しさ

PSySha が合成するプログラムの正しさは，プロファイル実行時と同じデータサイズ・スレッド数の実行時にのみ保証される．以下にその理由と，より一般的な場合に正しいプログラムが合成されやすくなるためのヒューリスティクスを述べる．

PSySha のプロファイル実行ではデータサイズとスレッド数に関しては具体的な値を用いるため，その具体的な値においてのみ正しく，一般には正しくない式を合成し得る．例えば 2 次元配列を扱うプログラムのプロファイル実行時に両次元の大きさが等しい配列を与えてしまうと，次元を混同したような添字式を合成する可能性がある．

一方，合成すべき添字式が満たす条件はプロファイル実行時に入力配列と共有メモリ上の配列で同じ値を検索することで求めている．そのため偶然同じ値を持つ配列要素がある場合には誤った添字式が合成される可能性がある．しかし，プロファイル実行時には全要素が異なる記号値で初期化した入力配列を用いるため値の混同は起きない．共有メモリへのコピー方法によっては同じ記号値が共有メモリ上の複数箇所にコピーされることはあるが，その場合には現在は最初に見つけた箇所を正解として合成を行なっている．正解の選び方によって条件を満たす添字式はとても複雑になる可能性がある．これに対処するには複数の正解を用いるように合成方法を拡張すべきだが，今後の課題としている．

なお，データサイズやスレッド数に関して記号値を用いない理由は，PSySha のプロファイル実行に用いている Kani-CUDA[16] がこれらについては具体的な数を与える設計になっているためである[4]．

完全なものではないが，次のような方法で任意のデータサイズ・スレッド数に対して正しいプログラムで正しいプログラムが合成されやすくなる．まずはプロファイル時に与えるデータサイズ，スレッド数を相異なる数としてカーネル起動することである．さらに，そのようなカーネル起動を複数通り行うことである．6 節の実験の範囲では前者のような設定のみで，正しいプログラムが合成されている．

またプログラムの正しさは現在は人手で丁寧に確認する必要がある．6 節の実験では合成されたプログラムを実際の入力データで実行し，最適化前の出力結果と比較することでも正しさを確認した．合成されたプログラムの正しさの検証を自動化することは今後の課題としている．

4.2.6　競合状態になるプログラムの合成

競合状態とは，複数のスレッドが同じメモリアドレスに同時に読み書きや書き込みをすることである．競合状態のプログラムは結果が一意にならない恐れがあり，GPGPU プログラムを記述する上で避けたい状態である．

[4]Kani-CUDA は CUDA のスレッド実行を Rosette プログラム合成フレームワーク上の逐次ループで模倣している．Rosette は繰り返し回数が記号値に依存するようなプログラムの扱いに大きな制限があるため，Kani-CUDA ではこれらを具体的な数に制限している．

実行フェーズで用いている Kani-CUDA は実行するプログラムが競合を起こす場合，エラーを発生させる機構を持つ．この機構により，最適化前のプログラムが競合を起こさないことが保証される．

しかし PSySha では 最適化前のプログラムが競合を起こさないが最適化後では競合を起こすようなプログラムを合成する場合がある．例えば，4.1 節で紹介したソースコード 4.2 の 7 行目のバリア同期が無い場合がそれに当てはまる．PSySha で合成されるプログラムが競合を起こさないことは保証されていないが，合成されたプログラムを Kani-CUDA で実行することによって，競合を起こすかどうかを確認することができる．また，Kani-CUDA は競合を起こすようなプログラムに対して，適切な位置にバリア同期命令を挿入する機能を持っている [17]．現在は，ユーザが上記のような後処理を施すことで合成されるプログラムの競合の不在を保証する．

また，Kani-CUDA の競合判定機構のために，結果が一意に定まるような正しいプログラムが合成できない場合がある．そのようなプログラムは同じメモリアドレスに同じ値を同時に読み書きする性質を持つ．例えば，後述するソースコード 4.8 である．このプログラムは結果は一意であるが，複数のスレッドが同じメモリアドレスに順不同で書き込む．10 行目や 21 行目の書き込みがそれに該当する．配列の添字式計算で `max` や `min` を用いているため，同じメモリアドレスに同じ値を同時に書き込んでいる．現在はソースコード 4.8 のようなプログラムを合成する場合は，競合検出機構を動作させない状態に切り替えて合成を行うことで対処する．

4.3 様々な共有メモリ最適化への対応

前節での合成はグローバルメモリから共有メモリにコピーする式が単純で共有メモリアクセス式が複雑になるような最適化に対する手法である．本節では異なるパターンの共有メモリ最適化に対するアプローチを示す．

4.3.1 例 1: 袖領域をコピーするような共有メモリ最適化

本節で袖領域を共有メモリにコピーするような最適化への合成器の対応を述べる．4.1 節で扱ったステンシルカーネルは共有メモリにコピーされない部分（袖領域）はグローバルメモリから取得していたために合成されるメモリアクセス式は複雑であった．一方で，袖領域を共有メモリにコピーする場合はメモリアクセス式の条件分岐が消え単純な記述になるが，一つのスレッドで複数の値をコピーする必要があるためコピーする式が煩

雑になる．ソースコード 4.4 は袖領域を共有メモリにコピーするような単純なステンシルカーネルである．

ソースコード 4.4: 袖領域を共有メモリにコピーするような最適化を行ったステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      int c2 = (threadIdx.y + 1) * (blockDim.x + 2) + threadIdx.x + 1;
6      __shared__ float sb[(blockDim.x + 2) * (blockDim.y + 2)];
7      sb[csb] = in[c];
8      if(threadIdx.x == 0){
9           sb[csb - 1] = in[(i == 0) ? c : c - 1]; }
10     if(threadIdx.x == blockDim.x - 1){
11         sb[csb + 1] = in[(i == nx - 1) ? c : c + 1]; }
12     if(threadIdx.y == 0){
13         sb[csb - blockDim.x - 2] = in[(j == 0) ? c : c - nx]; }
14     if(threadIdx.y == blockDim.y - 1){
15         sb[csb + blockDim.y + 2] = in[(j == 0) ? c : c - ny]; }
16     __syncthreads();
17     out[c] = sb[csb]
18         + sb[csb - 1]
19         + sb[csb + 1]
20         + sb[csb - blockDim.x - 2]
21         + sb[csb + blockDim.x + 2];
22 }
```

そこで袖領域をコピーするような最適化ではグローバルメモリから共有メモリにコピーする式を合成する．この場合の利用手順の一例を述べる．ソースコード 4.5 はソースコード 4.2 からユーザーが書き加える部分を示したものである．ユーザーは 5 行目から 7 行目の共有メモリの宣言や主要部のコピーを記述する．また合成される共有メモリへのコピー式が挿入される位置を示す `__insert()` を記述する．さらに 10 行目のような `__opt__` という記述をし，第一引数に書き換えたい共有メモリアクセス式，第二引数に書き換え前のメモリアクセス式を与える．この下描きを合成器に与えると合成器はソースコード 4.6 のように合成する．赤字部分が合成器によって記述される箇所である．

ソースコード 4.5: 合成器に入力する注釈付きプログラム

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      __shared__ float sb[(blockDim.x + 2) *(blockDim.y + 2)];
6      int csb = (blockDim.x + 2) *(threadIdx.y + 1) + threadIdx.x + 1;
7      sb[csb] = in[c];
8      __insert();
9      __syncthreads();
```

```
10      out[c] = __opt__(sb[csb - 1], in[(i == 0) ? c : c - 1])
11          + __opt__(sb[csb + 1], in[(i == nx-1) ? c : c + 1])
12          + __opt__(sb[csb - blockDim.x - 2], in[(j == 0) ? c : c - nx])
13          + __opt__(sb[csb + blockDim.x + 2], in[(j == ny-1) ? c : c + nx])
14          + sb[csb];
```

ソースコード 4.6: ソースコード 4.5 の合成結果

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2      int i = threadIdx.x + blockIdx.x * blockDim.x;
3      int j = threadIdx.y + blockIdx.y * blockDim.y;
4      int c = i + nx * j;
5      __shared__ float sb[(blockDim.x + 2) * (blockDim.y + 2)];
6      int csb = (blockDim.x + 2) * (threadIdx.y + 1) + threadIdx.x + 1;
7      sb[csb] = in[c];
8      if(threadIdx.x == 0){
9          sb[csb - 1] = in[(i == 0) ?  c :  c - 1];
10     }
11     if(threadIdx.x == blockDim.x - 1){
12         sb[csb + 1] = in[(i == nx - 1) ?  c :  c + 1];
13     }
14     if(threadIdx.y == 0){
15         sb[csb - blockDim.x - 2] = in[(j == 0) ?  c :  c - nx];
16     }
17     if(threadIdx.y == blockDim.y - 1){
18         sb[csb + blockDim.x + 2] = in[(j == ny - 1) ?  c :  c + nx];
19     }
20     __syncthreads();
21     out[c] = sb[csb - 1]
22         + sb[csb + 1]
23         + sb[csb - blockDim.x - 2]
24         + sb[csb + blockDim.x + 2]
25         + sb[csb]; }
```

続いて，この合成を実現するアルゴリズムについて説明する．合成アルゴリズムの概要は 4.2 節と同様である．合成する式のテンプレートは以下の通りである．

if(*cond*){(共有メモリアクセス式) = (グローバルメモリアクセス式)}

グローバルメモリから共有メモリのコピーをする式の両辺は__opt__の引数に与えられているメモリアクセス式を利用する．したがって，合成すべき部分は *cond* 部分の条件式である．プロファイル情報を記録する時に，__opt__の引数に与えた二つのメモリアクセス式の評価値が等しくないスレッドでコピーを行うような式を合成すればよい．そこでプロファイル時に各スレッドにおける次の真偽値を記録する．

$$(共有メモリアクセス式) == (グローバルメモリアクセス式) \quad (4.7)$$

(4.7) 式が偽となるスレッドで真となるような条件式を BExp から探索し，*cond* とすればよい．

4.3.2 例 2: テンポラルブロッキング最適化

本節ではテンポラルブロッキング最適化に対する PSySha の利用方法について述べる．テンポラルブロッキングはステンシル計算を対象とした共有メモリを用いた最適化である．2.1 節で述べた最適化は入力データをブロック分割しそれぞれを各ブロックの共有メモリ上で 1 反復分の更新値を求めることで高速化していた．テンポラルブロッキングではさらにブロックごとに数反復分の更新値を計算する．これによって，グローバルメモリから共有メモリへのロード回数が減り高速化が期待できる．

ソースコード 4.7 はソースコード 4.2 をテンポラルブロッキング最適化したプログラムである．このプログラムは 12 行目で 1 反復目の計算結果を共有メモリに書き込み，その後 24 行目で 2 反復目の計算結果を出力データに書き込む．

テンポラルブロッキングは最適化前と最適化後でメモリアクセス式の単純な置き換えではなく，プログラムの構造も大きく変わる．このような場合に合成器を利用したい場合は，まず共有メモリを使わずにプログラムの構造だけを変えたプログラムを合成器のユーザは記述する．共有メモリをデータを一時的に保存するために用いてなければいけない場合は，共有メモリの代わりにグローバルメモリを用いて記述する．

例えば，ソースコード 4.7 を合成したい場合は，ソースコード 4.8 の段階までユーザが手動で記述する．その上で `in` や `mid` へのアクセス式を共有メモリアクセス式に合成器を利用して置き換える．合成手順は 4.1 節で述べたものと同様である．

ソースコード 4.7: テンポラルブロッキング最適化されたステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *out, int nx, int ny){
2     int i, j, i2, j2, c, c2, csb, csb2;
3     i = blockIdx.x * (blockDim.x - 2) + threadIdx.x - 1;
4     i = max(0, i);
5     i = min(i, nx - 1);
6     j = blockIdx.y * (blockDim.y - 2) + threadIdx.y - 1;
7     j = max(0, j);
8     j = max(j, ny - 1);
9     c = nx * j + i;
10    __shared__ float sb[blockDim.x * blockDim.y];
11    csb = blockDim.x * threadIdx.y + threadIdx.x;
12    sb[csb] = in[(i == 0) ? c : c - 1]
13        + in[(i == nx - 1) ? c : c + 1]
14        + in[(j == 0) ? c : c - nx]
15        + in[(j == ny - 1) ? c : c + nx]
16        + in[c];
17    __syncthreads();
18    i2 = blockIdx.x * (blockDim.x-2) + min(threadIdx.x, blockDim.x-3);
19    i2 = min(i2, nx - 1);
```

```
20     j2 = blockIdx.y * (blockDim.y-2) + min(threadIdx.y, blockDim.y-3);
21     j2 = min(j2, ny -1);
22     c2 = nx * j2 + i2;
23     csb2 = blockDim.x * (j2%(blockDim.y-2) + 1) + i2%(blockDim.x-2) + 1;
24     out[c2] = sb[(i2 == 0) ? csb2 : csb2 - 1]
25         + sb[(i2 == nx - 1) ? csb2 : csb2 + 1]
26         + sb[(j2 == 0) ? csb2 : csb2 - nx]
27         + sb[(j2 == ny - 1) ? csb2 : csb2 + nx]
28         + sb[csb2];}
```

ソースコード 4.8: 共有メモリを使わずに記述されたテンポラルブロッキング最適化されたステンシルカーネル

```
1  __global__ void stencil_kernel(int *in, int *mid, int *out, int nx, int
        ny){
2      int i, j, i2, j2, c, c2, csb;
3      i = blockIdx.x * (blockDim.x - 2) + threadIdx.x - 1;
4      i = max(0, i);
5      i = min(i, nx - 1);
6      j = blockIdx.y * (blockDim.y - 2) + threadIdx.y - 1;
7      j = max(0, j);
8      j = max(j, ny - 1);
9      c = nx * j + i;
10     mid[c] = in[(i == 0) ? c : c - 1]
11         + in[(i == nx - 1) ? c : c + 1]
12         + in[(j == 0) ? c : c - nx]
13         + in[(j == ny - 1) ? c : c + nx]
14         + in[c];
15     __syncthreads();
16     i2 = blockIdx.x * (blockDim.x-2) + min(threadIdx.x, blockDim.x-3);
17     i2 = min(i2, nx - 1);
18     j2 = blockIdx.y * (blockDim.y-2) + min(threadIdx.y, blockDim.y-3);
19     j2 = min(j2, ny -1);
20     c2 = nx * j2 + i2;
21     out[c2] = mid[(i2 == 0) ? c2 : c2 - 1]
22         + mid[(i2 == nx - 1) ? c2 : c2 + 1]
23         + mid[(j2 == 0) ? c2 : c2 - nx]
24         + mid[(j2 == ny - 1) ? c2 : c2 + nx]
25         + mid[c2];}
```

第5章　実現

PSySha は主に次の二つのコンポーネントから構成されている．1) プロファイル情報を記録する機構を持つ CUDA プログラムの実行器．2) プロファイル情報から合成問題を生成し探索するソルバ．

1) は Kani-CUDA[16] の実行機を利用して実現した．Kani-CUDA は Rosette で CUDA プログラムから Rosette プログラムへの翻訳を記述することで実装されており，並列実行を模倣する実行モデルを持つ．プロファイル情報を記録する機構はメモリアクセス式の翻訳部分を一部改変するだけで実現できた．

2) は Java で実装し，コード数は約 1000 行である．

第6章　実験

本章ではPSyShaの性能を確認するために行った実験について述べる．

PSyShaが既存の合成器に比べ合成時間が削減できること，また，現実的なプログラムに対しても適用できることを検証するための実験を行った．いくつかのステンシル計算プログラムの最適化を対象にPSyShaとKani-CUDAを用いて合成し，その合成時間を計測した．

プログラムを合成する際，ユーザは最適化前のプログラムを実行する際に，入力データの大きさや記録する変数を指定する必要がある．合成を現実的な時間で完了するには入力データの大きさや記録する変数の数に限りがあることを検証するための実験を行った．姫野ベンチマークに対して，入力データの大きさと記録する変数の数をそれぞれ変化させて合成し，その合成時間を計測した．

また，PSyShaによって合成されたプログラムが最適化されていることを検証するためにの実験を行った．上述の実験で生成されたプログラムとCUDAで直接最適化したプログラムをGPU上で実行し計算性能を比較した．

これらの実験は表6.1，6.2の環境で行った．各実験について，プログラムの実行時間はシステムノイズを最小化するために，5回測定した中で最も短いものを記録した．

6.1　既存の合成器との合成時間の比較

提案する合成器PSyShaと既存の合成器Kani-CUDAの合成時間を比較を行った．対象は3次元熱拡散方程式プログラムと姫野ベンチマークプログラムそれぞれにおけるメモリアクセス式1つ[1]とし，最適化方針として，SHA(4.1節)，SHA-sleeve(4.3.1節)，TB(4.3.2節)の3つをそれぞれ行った．

プロファイル実行時の入力データサイズとスレッド数は次のように設定した．熱拡散方程式は入力データサイズが$9 \times 8 \times 3$で，1ブロックが3×4の12スレッド，3×2の6ブロックとした．姫野ベンチマークは入力デー

[1]複数のメモリアクセス式を最適化する場合には，各式を個別に最適化することで式数にほぼ比例する時間で合成できる[17]ため，1つの式に関する比較を行っている．

OS	macOS Mojave
CPU	2.5GHz Intel Core i7
メモリ	16GB 1600MHz DDR3
言語	Java version 1.8.0_144 Racket version 7.1
ソルバ	Z3 version 4.5.1

表 6.1: 合成器の実行環境

GPU	GP102 Titan Xp
CUDA コア数	3840 基
メモリ速度	11.4 Gbps
メモリ構成	12GB GDDR5X
メモリ帯域幅	547.7 GB/s
GPU 開発環境	CUDA 9.1

表 6.2: GPGPU プログラムの実行環境

タサイズが $17 \times 10 \times 3$ で，1 ブロックが 5×4 の 20 スレッド，3×2 の 6 ブロックとした．Kani-CUDA で記述した下描きや実験で用いた注釈付きプログラムは付録に添付する．合成時間の結果を表 6.3 に示す．また 300 秒以上かかった場合は打ち切りとし「> 300」と表記した．

PSySha はどの場合でも合成が完了し，合成時間は熱拡散方程式の SHA の場合は約 $\frac{1}{10}$，SHA-sleeve の場合は約 $\frac{1}{14}$ に削減できた．また姫野ベンチマークのような現実的なプログラムにおいても熱拡散方程式プログラムと近い時間で合成を完了することができた．合成結果はどれも正しい式が得られた．

6.2　入力データの大きさに対する合成時間

PSySha はプロファイル情報を記録する実行フェーズでは，小さな入力データに対して一度だけ実行し，複数の式に関する情報をそれぞれ生成する．入力データを小さくする目的は Kani-CUDA の実行時間の削減や，ソルバが解く制約を小さくして探索時間を削減である．この入力データの大きさはユーザが指示する．しかし，入力データが小さすぎたり，同じ値の定数を用いると，ソルバが局所解を出す恐れがある．ユーザはこのことを意識して入力データの大きさを決定する必要がある．

そこでこの実験ではユーザがどれだけストレスなく入力データの大きさを決定できるかという点で指標を示すために，1 つのプログラムに対し

	熱拡散方程式			姫野ベンチマーク		
	SHA	SHA-sleeve	TB	SHA	SHA-sleeve	TB
Kani-CUDA	42.9	27.5	> 300	> 300	> 300	> 300
PSySha	4.3	1.9	4.3	5.5	5.5	6.5

表 6.3: Kani-CUDA と PSySha の合成時間 (sec) の比較

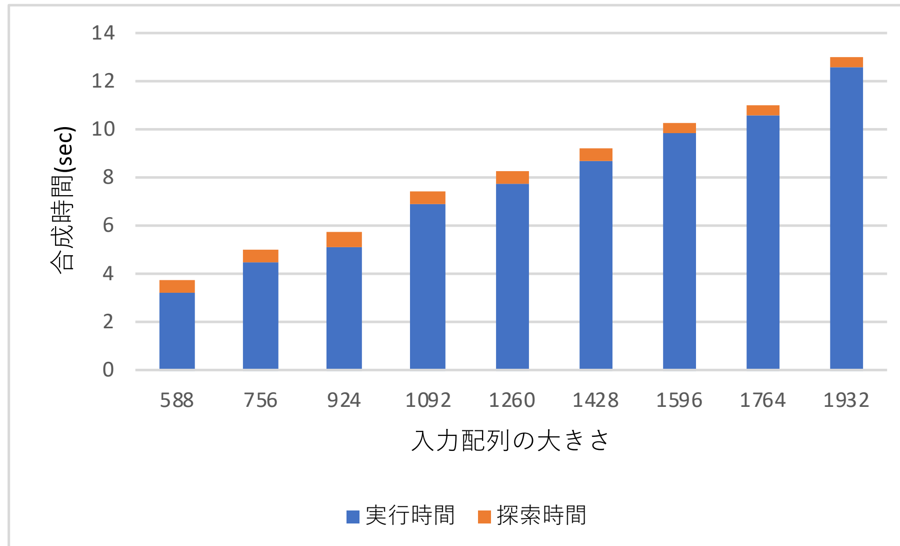

図 6.1: 入力データの大きさに対する姫野ベンチマークの合成時間 (sec)

て，入力データの大きさを変えて合成し，その時間を比較する．そこで前節と同様で一つのメモリアクセス式あたりの合成時間を示す．姫野ベンチマークに 4.1 節で紹介した最適化を施す．プロファイル情報として記録する変数の数は 6 個に固定して実験した．この 6 個の変数は今回のケースにおいて正しいプログラムを得る際に，必要最低限の変数らである．

実験結果を図 6.1 に示す．実行時間とはプロファイル情報を記録するために最適化前のプログラムの実行にかかる時間を表す．探索時間とはプロファイル情報から合成問題を抽出し，候補式集合の生成，探索までにかかった時間を表す．図から，入力データは実行時間へ大きく影響を与えることが読み取れる．また，実行時間はほとんど線形に増加している．今回のケースでは入力データの大きさを 1500 程度に設定することで，数秒での合成の完了を期待できることがわかった．入力データが大きい場合，記録する事例の数は増え，探索時にテストする事例の数は増えるが，今回のケースにおいて事例の数は探索時間に大きい影響を与えないことがわかった．

しかし，入力データの大きさが十分に大きいかどうかは最適化するプロ

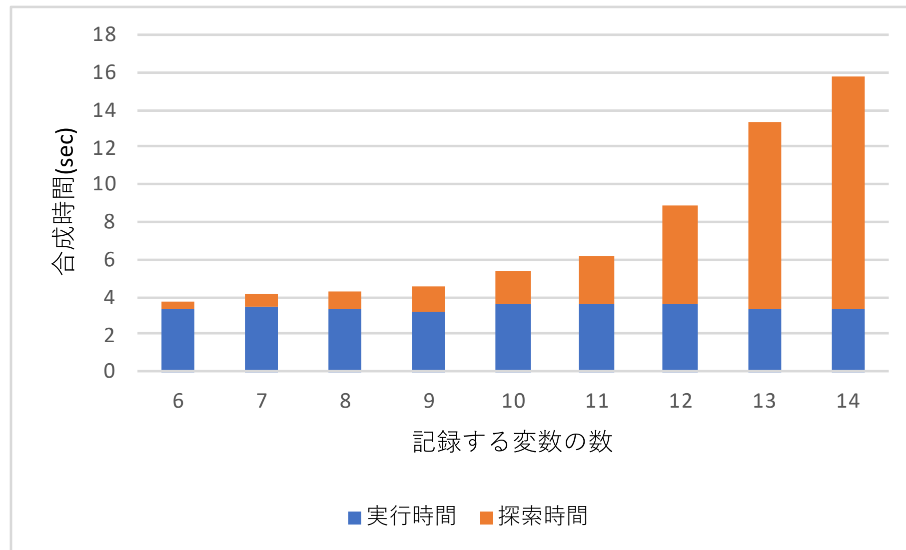

図 6.2: 記録する変数の数に対する姫野ベンチマークの合成時間 (sec)

グラムに依存するものである．そのため入力データの大きさをどの値に設定すべきかを設定することは難しい．

そこで Kani-CUDA の実行性能が CUDA の実行性能よりもどれだけ遅いかを比較した．入力データの大きさが 1932 でスレッド構造等を同様にして CUDA で実行したところ約 3×10^{-5} 秒で実行できた．つまり，Kani-CUDA は CUDA の実行時間の約 40 万倍の時間がかかる．現状の Kani-CUDA ではこれだけの実行性能しかないことをユーザは留意して入力データの大きさを検討する必要がある．

6.3　記録する変数の数に対する合成時間

PSySha のユーザは候補式の探索空間を削減するために，記録する変数を制限することができる．変数の数を制限すると，合成時間は短くなるが合成できる式の候補が減ってしまう．そのため場合によってはユーザは記録する式を注意して選択する必要がある．ユーザがどれだけストレスなく記録する変数を選択できるかを示すために，前節と同様に姫野ベンチの最適化を対象に，記録する変数の数を変えて合成時間を比較する．入力データの大きさは 588 に固定して実験を行した．

実験結果を図 6.2 に示す．前節でも述べたように図の横軸の最小である 6 個という数は，今回のケースで正しいプログラムを得る際に必要の変数らの最小の個数である．この 6 個以外に記録する変数らは正しい式を得る上では必要のない変数である．本実験においてソルバの探索を阻害するために故意に挿入している．図から，記録する変数の数は合成時間に大きく

影響与えることがわかる．探索時間について，最小二乗法による線形近似した際の R^2 値は 0.7885 で，指数近似した際の R^2 値は 0.9776 であった．今回のケースにおいては，記録する変数の数は 10 個前後以下に設定すると，数秒での合成の完了を期待できることがわかった．

6.4 合成されたプログラムの性能

PSySha が施す最適化の有効性を示すために，前節で紹介した最適化を施した姫野ベンチマークの性能について，PSySha で最適化したものと CUDA で直接共有メモリ最適化したものを比較した．Kani-CUDA で最適化したプログラムは手で記述した場合と同じプログラムが得られたため比較しない．

最適化手法について，共有メモリを更新領域と同じだけ確保し，メモリアクセス式に条件分岐を持たせる式は [7] を参考にした．一方，更新領域よりもひと回り大きく共有メモリを確保し，条件分岐を用いて袖領域をコピーする式は [11] で扱われた手法を用いた．本稿ではカーネルの最適化に焦点を当てるため，カーネルの処理性能のみを計測した．入力データの大きさが $256 \times 256 \times 512$ で 700 ステップ分の更新処理として計測した．図 6.3 が計測結果を表す．

PSySha で合成したプログラムは，どの最適化においても，CUDA で直接記述したものに相当する計算性能が出せた．SHA 最適化では，PSySha は CUDA で直接記述したプログラムと全く同じものが合成できたため，計算性能は等しい．SHA-sleeve 最適化と TB 最適化では，袖領域を共有メモリへコピーする式において異なるものが合成された．この違いを図 6.4 に示す．姫野ベンチマークは更新要素を求める際，更新要素が存在する二次元平面上の斜め方向の近傍要素を用いる．図 6.4 の (CUDA) と (PSySha) は，それぞれ CUDA で直接記述したプログラムと PSySha で合成したプログラムの 1 ブロックにおける斜め方向の袖領域のコピーの仕方を表している．(CUDA) は更新領域の左上の角の要素の更新値を求めるスレッドがコピーを行い，(PSySha) はコピーする要素と隣あっている要素の更新値を求めるスレッドがコピーを担当する．それぞれをコードで示したものが，ソースコード 6.1 とソースコード 6.2 である．それぞれ，コピーを行うスレッドが異なるため，`if` 文の数が変わる．そのためわずかに PSySha で合成したプログラムの性能が劣っている．

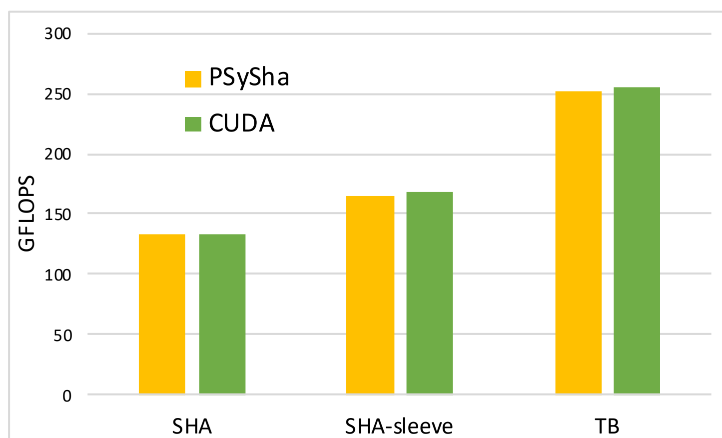

図 6.3: PSySha で合成した場合と CUDA で直接記述した場合の姫野ベンチマークプログラムの処理性能 (GFLOPS)

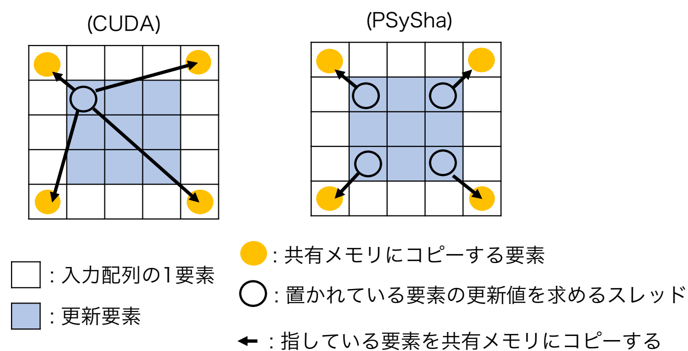

図 6.4: CUDA で直接記述した場合と PSySha で合成した場合の斜め方向の袖領域のコピーの仕方

ソースコード 6.1: CUDA で直接記述した姫野ベンチマークの共有メモリへのコピー式の一部

```
1  if(threadIdx.x == 0 && threadIdx.y == 0){
2      sb_m[0] = p[c-kmax-1];
3      sb_m[blockDim.x+2] = p[c-kmax+blockDim.x];
4      sb_m[(blockDim.y+1)*(blockDim.x+2)] = p[c+blockDim.y*kmax-1];
5      sb_m[(blockDim.y+2)*(blockDim.x+2)] = p[c+blockDim.y*kmax+blockDim.x
              ];
6  }
```

ソースコード 6.2: PSySha で合成された姫野ベンチマークの共有メモリへのコピー式の一部

```
1  if(threadIdx.x==0&&threadIdx.y==0){
2      sb1_m[csb-blockDim.x-3] = p[c-kmax-1];
3  }
4  if(threadIdx.x==blockDim.x-1&&threadIdx.y==0){
```

```
5      sb1_m[csb-blockDim.x-1] = p[c-kmax+1];
6  }
7  if(threadIdx.x==0&&threadIdx.y==blockDim.y-1){
8      sb1_m[csb+blockDim.x+1] = p[c+kmax-1];
9  }
10 if(threadIdx.x==blockDim.x-1&&threadIdx.y==blockDim.y-1){
11     sb1_m[csb+blockDim.x+3] = p[c+kmax+1];
12 }
```

第7章　関連研究

7.1　GPUDSL

高性能な GPGPU プログラムを容易に記述するためのデータ並列 DSL が研究されている．例えば Haskell 上の DSL である Accelerate や Ruby 上の DSL である Ikra がある．これらの DSL では，プログラマは map や reduce などのよく知られた並列スケルトンを用いてプログラムを記述する．GPUDSL は並列スケルトンによって記述されたプログラムから GPGPU プログラムを生成する．このとき，GPUDSL は GPU に特化した最適化を自動で施すため，ユーザは GPU を意識したプログラミングをする必要がないと言う利点がある．ただし，これらは共有メモリを用いた最適化は施されないものが多い．Ikra では reduce のみ共有メモリを用いた最適化が自動で施される．本研究はプログラマが共有メモリ最適化を施すことを援助する研究である．

7.2　逐次プログラムの自動並列化

逐次プログラムから，GPU 上で実行するためのホストコードとカーネルコードを自動生成する研究がされている．例えば，OpenACC[15] や PPCG[14] がある．これらのユーザは，逐次プログラムに対して，スレッド構造やブロック分割する際のブロックの大きさを指定するための指示文を挿入するだけで，並列化されたプログラムを得ることができる．その際，並列化されたプログラムの最適化はコンパイラが行うことに注意されたい．例えば，PPCG は逐次プログラム中のアフィン変換を行うループから多面体モデルを抽出することで，プログラムを解析し，並列プログラムを生成する．その際にプログラムの解析結果からグローバルメモリを共有メモリに割り当て，読み書きするコードも自動で生成する．本研究は合成器が共有メモリ最適化を半自動的に行うものであり，プログラマが共有メモリの利用の仕方を指示することができる．

7.3　GPGPU プログラム合成器

GPGPU を対象とする SyGuS を用いた支援には， OpenCL を対象とした SynthCL[13] や CUDA を対象とした Kani-CUDA[16] がある．これらは逐次プログラムや最適化されていないプログラムを仕様として与えると，それと同じ計算結果を返すプログラムを合成することができる．また SyGuS を採用しており，最適化されたプログラムのうち，記述が面倒な一部分だけを「穴」とするような下描きを与えれば，その穴を正しく埋めたプログラムを合成することができる．これらの合成器は汎用的な GPGPU プログラムを合成対象であるのに対し，本研究は共有メモリ最適化されたプログラムを合成対象としている．

第8章　結論と課題

8.1　結論

我々は最適化前のプログラムのプロファイル情報を利用して共有メモリ最適化を行う GPGPU プログラム合成器 PSySha を提案した．最適化前のプログラム中の変数やその値などの情報から，メモリアクセス式レベルの小さな合成問題を抽出し，独自に開発したソルバで解くことで実現した．

PSySha を用いて，3 次元熱拡散方程式プログラムと姫野ベンチマークに対していくつかの最適化を行った．共有メモリに袖領域をコピーする最適化，コピーしない最適化とテンポラルブロッキング最適化の 3 種類の最適化を対象として，どれも正しく合成できることを確認した．また，提案した合成器の合成時間の評価を行い，熱拡散方程式においては既存の合成器よりも $\frac{1}{10}$程度の 時間で合成を終えられることを確認した．さらに既存の合成器では一つのメモリアクセス式あたり，5 分以上の時間がかかっていた姫野ベンチマークに対しては一つのメモリアクセスあたり，6 秒以下の時間で合成できた．上記の最適化を対象に合成した姫野ベンチマークの性能評価を行い，合成されたプログラムが CUDA で直接記述した最適化済みプログラムに対して 2%以下のオーバーヘッドがあることを確認した．

8.2　今後の課題

本研究において，PSySha はステンシル計算のみを対象として合成実験を行った．異なる種類の GPGPU プログラムに対して有効性の検証をすることは今後の課題である．例えば，地震伝播シミュレーションなどで用いられる畳み込み演算などである．

PSySha は算術式と論理式の探索空間を生成し探索している．算術式に関しては非常に単純なアルゴリズムで生成しているため，等価である式が複数含まれるような冗長な探索空間を生成している．より最適な探索空間を生成することは今後の課題である．一方，論理式に関しては非常に限られた探索空間を生成している．今回の実験においてはどのプログラムも合

成できたが今後，合成できないようなプログラムが発見された時は改善すべき課題である．

参考文献

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E. and Udupa, A.: Syntax-guided Synthesis, *Proceedings of Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8 (2013).

[2] 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*, DAMP '11, New York, NY, USA, ACM, pp. 3–14 (2011).

[3] Gulwani, S.: Dimensions in Program Synthesis, *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, New York, NY, USA, ACM, pp. 13–24 (2010).

[4] Holewinski, J., Pouchet, L.-N. and Sadayappan, P.: High-performance Code Generation for Stencil Computations on GPU Architectures, *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, New York, NY, USA, ACM, pp. 311–320 (2012).

[5] Hong, S. and Kim, H.: An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness, *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, New York, NY, USA, ACM, pp. 152–163 (2009).

[6] Kamil, S., Cheung, A., Itzhaky, S. and Solar-Lezama, A.: Verified Lifting of Stencil Computations, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, New York, NY, USA, ACM, pp. 711–726 (2016).

[7] Maruyama, N. and Aoki, T.: Optimizing Stencil Computations for NVIDIA Kepler GPUs, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pp. 89–95 (2014).

[8] Masuhara, H. and Nishiguchi, Y.: A Data-Parallel Extension to Ruby for GPGPU: Toward a Framework for Implementing Domain-Specific Optimizations, *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE '12, New York, NY, USA, ACM, pp. 3–6 (2012).

[9] Munshi, A., Gaster, B., Mattson, T. G., Fung, J. and Ginsburg, D.: *OpenCL Programming Guide*, Addison-Wesley Professional, 1st edition (2011).

[10] NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2010). Version 3.2.

[11] Phillips, E. H.: Implementing the Himeno benchmark with CUDA on GPU clusters, *Proceedings of International Symposium on Parallel and Distributed Processing (IPDPS), 2010*, pp. 1–10 (2010).

[12] Rogers, T. G., O'Connor, M. and Aamodt, T. M.: Divergence-aware Warp Scheduling, *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, New York, NY, USA, ACM, pp. 99–110 (2013).

[13] Torlak, E. and Bodik, R.: A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 49, No. 6, pp. 530–541 (2014).

[14] Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C. and Catthoor, F.: Polyhedral Parallel Code Generation for CUDA, *ACM Trans. Archit. Code Optim.*, Vol. 9, No. 4, pp. 54:1–54:23 (2013).

[15] Wienke, S., Springer, P., Terboven, C. and an Mey, D.: OpenACC: First Experiences with Real-world Applications, *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, Berlin, Heidelberg, Springer-Verlag, pp. 859–870 (2012).

[16] 蟹暁, 朝倉泉, 増原英彦, 青谷知幸: バリア同期と共有メモリを備えたGPGPU プログラム合成器 Kani-CUDA, 情報処理学会第 113 回プログラミング研究会 (2017).

[17] 蟹暁, 朝倉泉, 増原英彦, 青谷知幸: GPGPU プログラム最適化のためのプログラム合成器とその適用手法, cross-disciplinary workshop on computing System, Infrastructures, and programminG (xSIG) (2018).

付 録 A 実験で用いたプログラム

ソースコード A.1: 最適化前の 3 次元熱拡散方程式プログラム

```
__global__ void diffusion_kernel(float *in,
                                 float *out,
                                 int nx, int ny, int nz,
                                 float ce, float cw, float cn, float cs,
                                 float ct, float cb, float cc) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;
  int c = i + j * nx;
  int xy = nx * ny;
  for (int k = 0; k < nz; ++k) {
    int w = (i == 0)      ? c : c - 1;
    int e = (i == nx-1)   ? c : c + 1;
    int n = (j == 0)      ? c : c - nx;
    int s = (j == ny-1)   ? c : c + nx;
    int b = (k == 0)      ? c : c - xy;
    int t = (k == nz-1)   ? c : c + xy;
    in[c] = cc * in[c] + cw * in[w] + ce * in[e] + cs * in[s]
        + cn * in[n] + cb * in[b] + ct * in[t];}
 }
```

ソースコード A.2: ソースコード A.1 の最適化 (SHA) を施す下描き

```
__global__ void diffusion_kernel(float *in,
                                 float *out,
                                 int nx, int ny, int nz,
                                 float ce, float cw, float cn, float cs,
                                 float ct, float cb, float cc) {
  // 赤字部分がユーザが記述する部分
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;
  int c = i + j * nx;
  int xy = nx * ny;
  __shared__ float sb[blockDim.x*blockDim.y];
  int csb = threadIdx.x + threadIdx.y *blockDim.x;
  for (int k = 0; k < nz; ++k) {
    sb[csb] = in[c];
    __syncthreads();
    int w = (i == 0)      ? c : c - 1;
    int e = (i == nx-1)   ? c : c + 1;
    int n = (j == 0)      ? c : c - nx;
    int s = (j == ny-1)   ? c : c + nx;
    int b = (k == 0)      ? c : c - xy;
```

```
21     int t = (k == nz-1)   ? c : c + xy;
22     in[c] = cc * in[c]
23     // (?  a b ...) はその部分が a, b,... のいずれかで埋められるという文法を表す
24      + cw * ((threadIdx.(?  x y)==(?  0 (blockDim.(?  x y)-(?  0 1))))
25             && i != 0)
26              ?  sb[(i (?  == !=) (?  0 ((?  nx ny) - (?  0 1))))
27                   ?  csb
28                   :  csb (?  + -) (?  1 (blockDim.(?  x y)))]
29              : in[w]
30      + ce * in[e] + cs * in[s]
31      + cn * in[n] + cb * in[b] + ct * in[t];}
32   }
```

ソースコード A.3: ソースコード A.1 の最適化 (SHA-sleeve) を施す下描き

```
1  __global__ void diffusion_kernel(float *in,
2                                  float *out,
3                                  int nx, int ny, int nz,
4                                  float ce, float cw, float cn, float cs,
5                                  float ct, float cb, float cc) {
6    // 赤字部分がユーザが記述する部分
7    int i = blockDim.x * blockIdx.x + threadIdx.x;
8    int j = blockDim.y * blockIdx.y + threadIdx.y;
9    int c = i + j * nx;
10   int xy = nx * ny;
11   __shared__ float sb[(blockDim.x+2)*(blockDim.y+2)];
12   int csb = 1+threadIdx.x+(threadIdx.y+1) *(blockDim.x+2);
13   for (int k = 0; k < nz; ++k) {
14     int w = (i == 0)      ? c : c - 1;
15     int e = (i == nx-1)   ? c : c + 1;
16     int n = (j == 0)      ? c : c - nx;
17     int s = (j == ny-1)   ? c : c + nx;
18     int b = (k == 0)      ? c : c - xy;
19     int t = (k == nz-1)   ? c : c + xy;
20     sb[csb] = in[c];
21     __syncthreads();
22     if((?  i j threadIdx.(?  x y))
23            (?  == !=)(?  0 (blockDim.(?  x y)-(?  0 1)))){
24         sb[csb-1] = in[w];
25     in[c] = cc * in[c]
26      + cw *sb[csb-1]
27      + ce * in[e] + cs * in[s]
28      + cn * in[n] + cb * in[b] + ct * in[t];}
29   }
```