修士論文

題目

アセンブリコードと
ソースコードの解析を併用した
GPUプログラムの性能予測

指導教員

大野 和彦 講師

2018年

三重大学大学院 工学研究科 情報工学専攻
コンピュータソフトウェア研究室

中井 裕登(416M515)

内容梗概

グラフィックス処理用プロセッサである GPU に汎用的な演算を行わせる GPGPU は，CPU 以上の計算性能を発揮することもあり，近年，期待が高まっている．GPU は多数のスレッドを並列に実行できるが，高速化にはハードウェアの特性を意識したコーディングが求められる．特にメモリ上のデータレイアウトはプログラムの性能への影響が大きく最適化が必要である．しかし，アーキテクチャに関する専門的な知識を求められるため，自動最適化が望まれている．

最適なデータレイアウトの選択に必要な情報を取得する方法として GPU 用のソースコードである CUDA コードに対する静的解析や実行時情報を利用する動的解析がある．CUDA コードの静的解析では特定のプログラムに対して最適なレイアウトを決定できない可能性がある．例えば，コンパイラによる最適化によって複数のメモリアクセス命令が 1 つにまとめられる場合である．このとき，CUDA コードに対する静的解析ではそれを検出できないため，最適なレイアウトを選択できない．一方で動的解析は実際にプログラムを実行するため，静的解析より高い精度を持つが膨大な解析時間を必要とする．

そこで本研究では，プログラムに最適なデータレイアウトの選択を目的とした GPU プログラムの性能予測手法を提案する．本手法では CUDA コードの解析と GPU アセンブリコードである PTX コードの解析を併用する．CUDA コードに対して静的解析を行うことで制御構造とメモリアクセスパターンを抽出し，それを基にプログラムの制御構造とその箇所における実行時間を表す重み付き制御フローグラフを生成する．そして PTX コードを参照し，実際に実行される命令列を取得する．この命令列を利用することでグラフの各ノードの実行時間を予測する．性能評価の結果，従来手法では性質の異なるプログラム 5 本中 3 本で最適なレイアウトを選択できなかったのに対し，提案手法ではそれらにおいて最適なレイアウトを選択できた．

Abstract

GPGPU(General Purpose computing on Graphics Processing Units) gets attention from various fields because of high computational performance. GPU can execute many threads in parallel. However the users must devote coding effort to optimize GPGPU program using CUDA. Especially, the data layout on memory affects the performance of programs thus optimization is necessary. However it is difficult for users to find the optimal layout. Thus automatic optimization is in demand.

Static analysis on CUDA code(source code for GPGPU) and dynamic analysis are method to obtain the information using determine the optimal layout. When a few memory accesse instructions are combined by compiler, the analysis can not detect the combine. Thus using static analysis on CUDA code may determine the layout which is not optimal. dynamic analysis is more accurate than static analysis. However the analysis need amount of time.

In this research, we propose a method which estimates the execution time of GPU programs for determining the optimal data layout. We use static analysis on CUDA code and PTX code(GPU-assembly code). Using static analysis on CUDA code, we extract the control structure and memory access pattern. Based on the control structure, we generate control flow graph with cost which shows the control structure and the elapsed time of program. To obtain the instructions which are executed actually on GPU, we also refer to PTX code. Using the instructions and memory access pattern, we estimate the cost of each node in the graph. We evaluated with 5 programs which have different character. As the result of evaluation, the previous method could not determine optimal layout with 3 programs. In contrast, the proposed method could determine optimal layout with them.

目 次

図 目 次

表 目 次

1　はじめに

グラフィックス処理用プロセッサである GPU(Graphics Processing Unit) に汎用的な演算を行わせる GPGPU(General Purpose computation on GPUs) は，CPU 以上の計算性能を発揮することから期待が高まっている [1,2]. GPU は多数のスレッドを並列に実行できるが，高速化にはハードウェアの特性を意識したコーディングが求められる．

メモリ上のデータレイアウトは，プログラムの性能への影響が大きいため最適化が必要である．さらに，一般にデータ構造はプログラマにとってプログラムの理解が容易となるように記述されるため，メモリアクセスの時間的局所性が考慮されていない場合がある．しかし，アーキテクチャについての専門的な知識を求められるためプログラマが最適なレイアウトを見つけることは困難であり，自動最適化が望まれている．

既存手法としてメモリ上のデータレイアウトの最適化に必要な情報を CUDA コードに対する静的解析により取得するものがある [8,9]．しかし，特定のプログラムに対して最適なレイアウトを決定できない可能性がある．例えば，コンパイラによる最適化によってメモリアクセス命令の実行回数が変わる場合である．また，間接参照を用いるものや，実行時の条件分岐によって配列の添え字式の値が変わるプログラムは実行するまでメモリアクセスパターンが不明であるため対応できない．

そのため，動的解析によってメモリアクセス命令の実行回数とアクセス先のアドレスを取得する手法が存在する [7, 10]．これはプログラムの規模に比例して情報取得のためのオーバーヘッドが大きくなる．さらに，入力データによって挙動が変わる場合は解析結果による最適化の効果が保証されない．

本研究では，プログラムに最適なデータレイアウトの選択を目的として GPU プログラムの性能予測手法を提案する．本手法では CUDA コードに加えて GPU アセンブリコードへの解析も行う．CUDA コードに対して静的解析を行うことでカーネル関数の制御構造とメモリアクセスパターンを抽出し，それを基にプログラムの制御構造とその箇所における実行時間を表す重み付き制御フローグラフを生成する．そして GPU アセンブリコードである PTX コードを参照し，実際に実行される命令を取得する．これを用いることでグラフの各ノードの実行時間を予測する．

以降，まず 2 章で研究の背景として CUDA と GPU アーキテクチャ，データレイアウトの最適化について解説する．続く 3 章で提案手法を解説し，4 章で提案手法による各レイアウトに対する予測実行時間と実測し

た実行時間を比較した結果を示す．そして，5 章で GPU プログラムにおけるデータレイアウト及びメモリアクセスの自動最適化に関する関連研究を紹介する．最後に，6 章で本論文をまとめる．

2　背景

2.1　GPU と CUDA

2.1.1　GPU

GPU は演算を行うコアを大量に搭載し多数の処理を並列に実行できる．GPU ではコア数を超えるスレッドを生成でき，これらの大量のスレッドは 32 スレッド単位で分割され，管理・実行される．この 32 スレッドのグループをワープという．ワープ内の 32 スレッドは同じ命令を実行する SIMD 型の並列処理を行う．

GPU はキャッシュを搭載した階層型のメモリアーキテクチャを採用している．GPU のメインメモリであるデバイスメモリへのアクセスは L2 キャッシュのラインサイズである 128byte 単位で行われる．以後，本論文におけるキャッシュとは L2 キャッシュを指すものとする．ワープ内のスレッドが同時に同一キャッシュライン上のデータにアクセスすれば，複数のデータ転送を一度のデバイスメモリへのアクセスで行える．このようなアクセスをコアレッシングアクセスという．また，同一ライン内のデータに対して，時間的局所性のあるアクセスを行えば，キャッシュメモリ上にデータが存在するので高速にアクセスできる．

2.1.2　CUDA

CUDA [2] は nVIDIA 社が提供するコンパイラ・ライブラリを含めた GPGPU 統合開発環境であり, ユーザは C/C++を拡張した文法とライブラリ関数を用いて CUDA プログラムを開発する．CUDA プログラミングモデルを図 2.1 に示す．CUDA において, CPU 側はホスト, GPU 側はデバイスと呼ぶ．デバイスは PCI-Express を通じてホストにより制御され, ホストから与えられる計算処理を数千個の CUDA コアで並列実行する．このコアを搭載する演算部をストリーミングマルチプロセッサ (SM) という．ホスト・デバイスの各 CPU コア・CUDA コアは図 2.1 に示すように, 自身が接続するホストメモリ・デバイスメモリにのみそれぞれアクセスする．ホストメモリ・デバイスメモリ間のデータ転送はユーザ自身が CUDA ライブラリ関数を用いて記述する必要がある．CUDA では低レベルなコーディングがサポートされており，データアクセスやスレッドマッピングの最適化など，GPU アーキテクチャを意識したプログラミングによるチューニングが可能である．

CUDA にはビルトイン変数が存在し，宣言なしにカーネル関数内で使用できる．各ブロック・スレッドにはそれぞれ番号が割り振られており，gridDim.x でブロックの個数を，blockIdx.x でブロック番号（gridDim.x）を，blockDim.x でスレッドの個数を，threadIdx.x でスレッド番号（blockDim.x）を，それぞれ得ることができる．上で示した変数では x 方向についての値を得ているが，.x の部分を .y，.z とすることでそれぞれ y 方向と z 方向の値を得ることができる．ブロックの番号はユニークであるがスレッド番号はブロックごとに割り振られているため，カーネル関数を起動したとき全スレッドで見るとブロックの数だけ同じ番号が重複してしまう．式 blockDim.x × blockIdx.x ＋ threadIdx.x の値は各スレッドごとにユニークであり，0 から始まる連続した値となる．よってここではこの式の値をスレッドの ID として用いることとし，以下 *tid* で表す．

また，CUDA コンパイラは自動で最適化を行うため，命令の実行順や命令数が変わることがある．コンパイル時にオプション（--ptx）を付けることで CUDA コンパイラは GPU アセンブリ言語で記述された PTX コード生成する．これを参照することで実際に GPU 上で実行される命令がわかる．

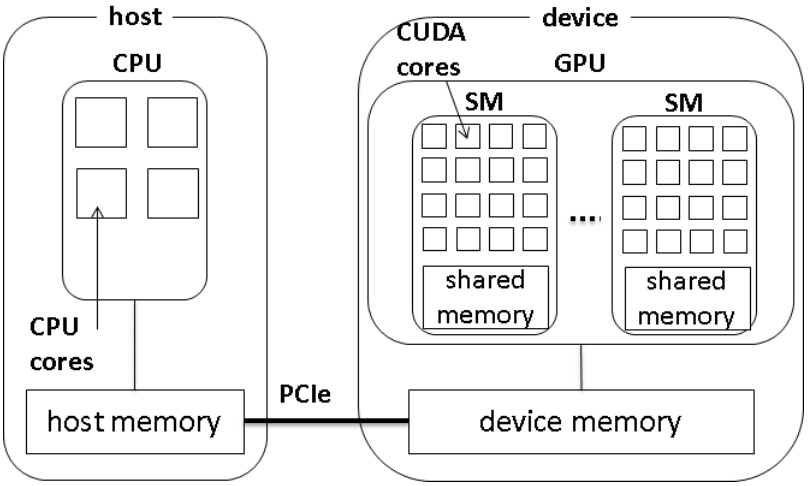

図 2.1: GPU アーキテクチャ

2.1.3　ワープスケジューリング

SM 内のスケジュールの単位はワープである．ワープ内のすべてのスレッドは 32 個の CUDA コア上で同じ命令を並行して実行する．異なるワープは独立して実行することができるため，ワープスケジューラは使用可能な空き CUDA コアがある場合に，あるワープがストールした際にワープを切り替える．そして，別のワープが命令を実行することで図 2.2 の着色部分のようにメモリアクセス命令のレイテンシを隠蔽することができる．

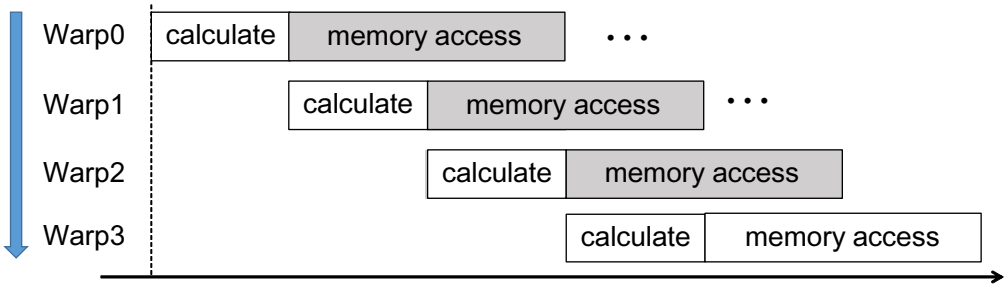

図 2.2: ワープスケジューリング

2.2　データレイアウトの最適化

2.2.1　AoS と SoA

構造体の配列 (Array Of Structure) と配列の構造体 SoA(Structure Of Array) の定義の例を図 2.3 に示す．また，このときメモリ上では図 2.4 のように配置される．以下では，構造体の配列を AoS，配列の構造体を SoA と表記する．各スレッドが配列の各要素を処理対象とする場合，各メンバを参照したときのアクセス先は図 2.4 の着色部分になる．この性質により，連続した領域へ同一ワープ内のスレッドがアクセスするとコアレッシングアクセスの効果が大きくなる．しかし図 2.4 のように AoS の特定メンバを一斉にアクセスすると，不連続領域へのアクセスとなる．そこで，AoS を SoA に変換することで連続した領域へのアクセスとなり，このようなメモリアクセスを高速化できる．だが，メモリアクセスパター

```
struct AOS{
    int x, y, z;
};

Struct AOS aos[N];
```

```
struct SOA{
    int x[N], y[N], z[N];
};

Struct SOA soa;
```

図 2.3: AoS と SoA の定義

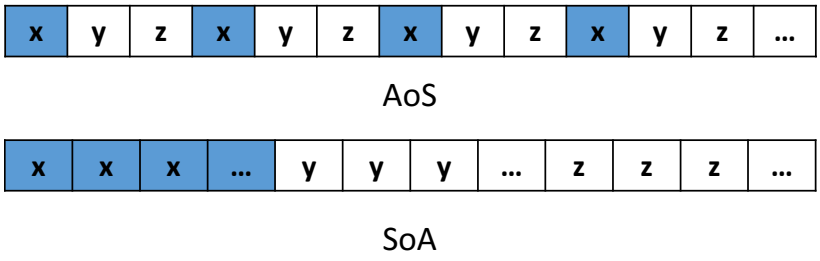

図 2.4: AoS と SoA のメモリ上の配置

ンによっては AoS の方が高速となることもあるため，プログラムに適したレイアウトを選択する必要がある．

2.2.2 Array-of-Structure-of-TiledArrays(ASTA)

一般に高速とされる SoA に代わるレイアウトとして Sung ら [14] が最適化の選択肢に取り入れたものがタイル化 AoS(ASTA) である．定義の例を図 2.5 に示す．また，このときメモリ上では図 2.6 のように配置される．このレイアウトは各構造体メンバがタイル数ずつ配置されることが特徴である．図 2.6 はタイル数を 4 にしたときの例である．このタイル数によって性能が変わるため，プログラムに適したタイル数を設定する必要がある．これにより，AoS が持つ空間的局所性への優位性と SoA が持つ連続アクセスによる優位性を両立している．しかし，Sung ら [14]，Kofler ら [8] の NVIDIA GPU を用いた評価では SoA と同程度の性能となっており，AoS，SoA より優れるとはいえない．

```
struct ASTA{
   int x[4], y[4], z[4];
};

Struct ASTA asta[N/4]
```

図 2.5: ASTA の定義

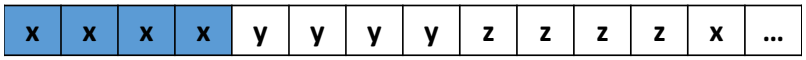

図 2.6: ASTA のメモリ上の配置

2.2.3 構造体のアライメントによる最適化

GPU の各コアによるデバイスメモリへの書き込み，読み出しは，1，2，4，8，16byte 単位でのアクセス命令のいずれかにより実行される．デバイスメモリへの 4byte 変数の書き込みは，4byte 単位の書き込み命令によって実行される．4byte メンバを 2 個以上持つような構造体の値のデバイスメモリへの書き込みも，4byte 単位の書き込み命令を 2 回実行する．このとき，8byte や 16byte 単位の書き込み命令によって複数のメンバの書き込みや読み出しを 1 度の命令で実行するためには，構造体をアライメントする必要がある．CUDA プログラミングでは構造体のアライメントをサポートしており，__align__キーワードによって適用できる．アライメント後の構造体の配列を図 2.7 に示す．

構造体のアライメントにより，構造体変数への書き込み・読み出しが効率よく行われる．たとえば，4byte のメンバを 4 個持つ構造体を 16byte でアライメントした場合，デバイスメモリへの書き込みは 16byte 書き込み命令 1 回で実行される．4byte のメンバを 7 つ持つ構造体を 16byte でアライメントした場合，16byte 単位の書き込み 1 回，8byte 単位の書き込み 1 回，4byte の書き込み 1 回によって実行される．このようにアライメントにより複数ワードの書き込みまたは読み出しを 1 命令で実行することにより，アクセスを効率化できる．

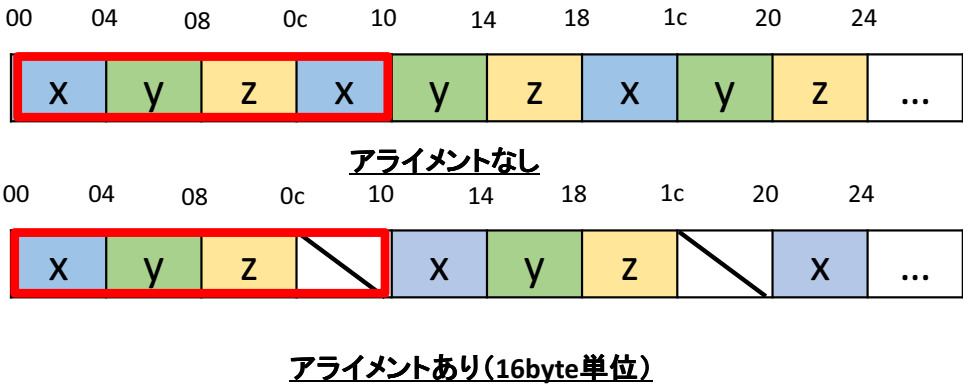

図 2.7: アライメントを適用した構造体のメモリ上の配置

3　提案手法

本稿では，データレイアウトの自動最適化を目的とするGPUプログラムの性能予測手法を提案する．

AoSではアライメントによって複数のメモリアクセス命令が1つにまとめられるため，CUDAコードから予想される命令数と実際に実行される命令数が異なる．また，コンパイラによって演算の実行順序も変更される可能性がある．このような場合，従来のようにCUDAコードの解析のみを利用する手法では予測性能と実際の性能に誤差が生じ，それが原因となって最適なレイアウトの選択を誤る可能性がある．

そこで，CUDAコードの解析とGPUアセンブリコードであるPTXコードの解析を併用する．図3.8に提案手法の全体図を示す．CUDAコードの解析によってカーネル関数の制御構造とメモリアクセスパターンを，PTXコードの解析によって実際に実行される命令とループのアンロール情報を，それぞれ取得する．これらを基に制御フローグラフを拡張した重み付き制御フローグラフを生成する．このグラフを用いてコアレッシングアクセスやキャッシュヒット、メモリアクセス命令のオーバーラップを考慮することでワープ当たりの実行時間を予測する．

カーネル関数における性能をより正確に予測する場合はカーネル実行の開始から終了までの時間を予測する必要がある．あるワープではAoSがSoAより実行時間が小さいが，別のワープではAoSがSoAより実行時間が大きくなる場合も考えられる．しかし，ワープ当たりの平均実行時間でAoSがSoAが優れる場合に，カーネル実行時間でSoAがAoSより優れることはない．そのため，本手法ではレイアウト選択にワープ当たりの平均実行時間を予測する．

3.1　本手法における定義及び制限事項

以下に本論文で用いる用語を定義する．本手法ではメモリアクセス命令と演算命令の実行時間を予測する．メモリアクセスについては，ホストからデバイスに転送された配列へのアクセスのみを対象とする．この配列はデバイスメモリであるDRAMに割り当てられるため，アクセスレイテンシが大きいのに対し，ローカル変数はレジスタに割り当てられるためアクセスレイテンシが小さいからである．本手法では同一ワープ内スレッドのアクセス先を得るために配列のインデックス式を静的解析する．

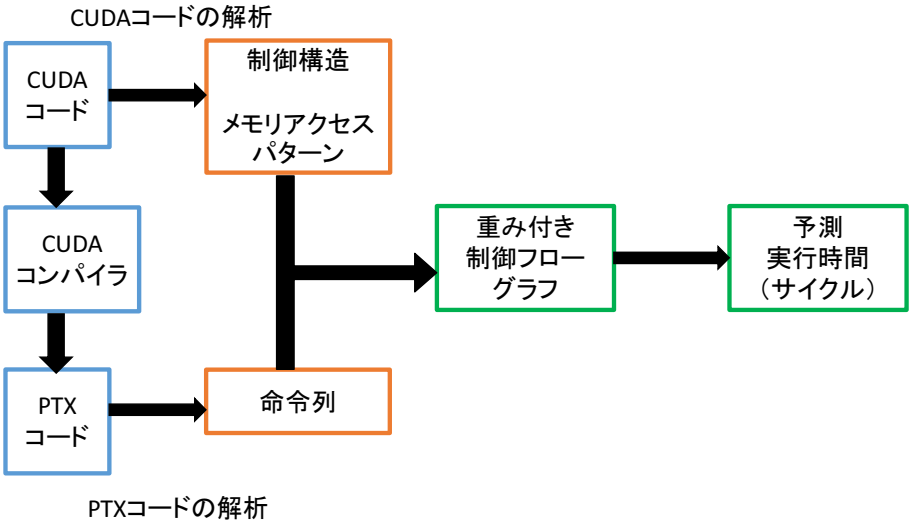

図 3.8: 提案手法の全体図

- ターゲット変数
  コード上の k 重ループの中で配列へのアクセスが n 回検出されるとする．各インデックス式 $e_0, \ldots, e_{n-1}$ において，ビルトイン変数 tid と全てのループ変数 $i_0, \ldots, i_{k-1}$ をターゲット変数と見なす．

- インデックス式の正規形
  インデックス式 e の正規形を $N(e)$ として記述し以下の式で表す．

$$\begin{aligned} N(e) &= (C_0^I \times tid \\ &+ (C_0^L \times i_0 + \ldots + C_{k-1}^L \times i_{k-1}) \\ &+ C \end{aligned}$$

  正規形はターゲット変数の項が一次である多項式でなければならない．C_p^I と C_q^L は各項の係数であり，tid，C と共にループ内不変である．

- 重み付き制御フローグラフ
  制御フローグラフを拡張したものである．このグラフは命令文ノードと制御文ノードを持つ．1 つの命令ノードは PTX 上の繰り返し文

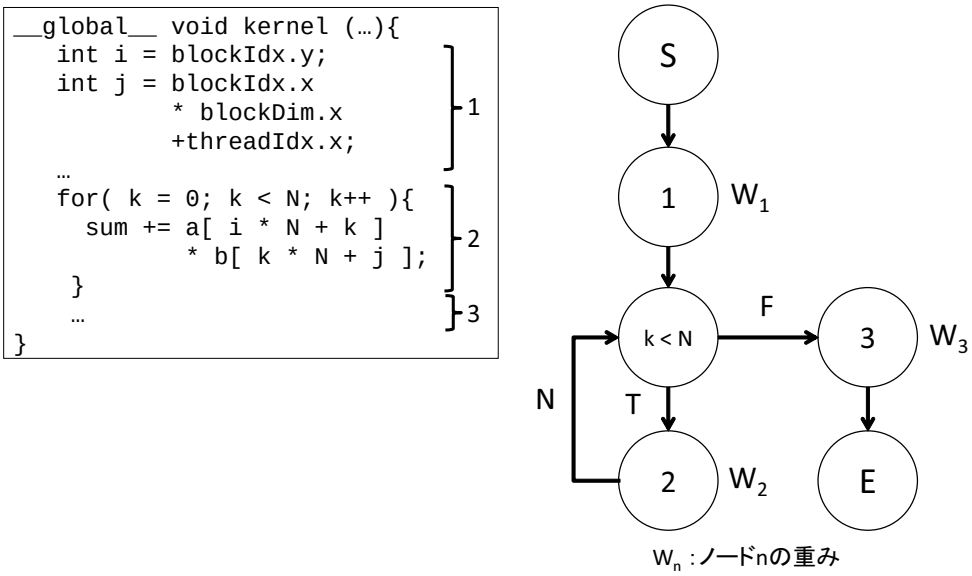

図 3.9: サンプルコードと重み付き制御フローグラフ

を含まない連続した命令列とそのノードの重みを持つ．例として図 3.9 (a) から抽出した制御構造を基に生成したグラフを図 3.9 (b) に示す．ここでの重みはノードが持つ命令列のレイテンシの和である．本手法では計測によって取得したレイテンシ [15] を用いる．制御文ノードは繰り返し文の条件式を持つ．各命令文ノードの重みを求め，総和を取ることでカーネル関数の実行時間を予測することができる．このときループボディに対応する命令文ノードの重みをループ回数倍することでループを考慮する．

カーネル関数は下記の条件を満たすものとする．

1. 全てのループ回数は固定でコンパイル時に分かっている．
2. 全ての配列インデックス式は正規形に変形できる．

現状，不定回ループと非線形なインデックス式は対応していないが，多くのカーネル関数は解析できる．これは，GPU プログラムでは不規則なアクセスパターンは非効率的で避けられる傾向にあるためである．条件を満たさないインデックス式によるアクセスは実行時間予測の対象から除外する．

また，制御文においては繰り返し文である for と while を検出し，上述のように実行時間の予測に反映させる．if や switch といった条件分岐については検出しない．これは静的解析では条件文が真となる回数を取得できないからである．しかし GPU ではワープ内のスレッドが異なる分岐パスを選択した場合，真になるパスと偽になるパスの両方を実行するため実行効率が下がる．そのため分岐を減らす，もしくは使用しない傾向があるため，多くのカーネル関数の実行時間は問題なく予測可能である．

3.2 CUDA コードの解析

3.2.1 制御構造の抽出

重み付き制御フローグラフを生成するためにカーネル関数の制御構造を抽出する．カーネル関数が記述された CUDA コードに対して静的解析を行い，繰り返し文を検出する．

3.2.2 メモリアクセスパターンの解析

キャッシュヒットとコアレッシングアクセスの効果を予測するためにメモリアクセスパターンを抽出する．このとき CUDA コード上のカーネル関数における配列のインデックス式に注目し，配列へのアクセス時に発生するメモリトランザクション数を求めることでメモリアクセスを分類する．メモリトランザクション数を求めるアルゴリズムを図 3.10 に示す．また，アルゴリズム内で使用する関数と変数を以下で定義する．

- *transform()* : 引数として与えたインデックス式を正規形に変換する式中の非ターゲット変数はインデックス式の登場時に置き換え可能ならば置換する．例えば，*tid* などがローカル変数に代入されており，それを検出可能な場合である．
- *calc_index()* : 引数として与えたスレッド ID をビルトイン変数に代入したときのインデックス式の正規形を求める
- *element_size* : アクセスする配列の 1 要素当たりのサイズ
- *size_per_warp* ：ワープ当たりのアクセスデータサイズ

1: **for all** インデックス式 **in** カーネル関数 **do**
2: 　正規形 ← *transform(*インデックス式*)*
3: **end for**
4: **for all** 正規形 **do**
5: 　e_{tid} ← *calc_index(tid)*
6: 　e_{tid+1} ← *calc_index(tid+1)*
7: 　*diff* ← $|e_{tid} - e_{tid+1}|$
8: 　*stride* ← *diff* × *element_size* //アクセスストライドを求める
9: 　**if** *stride* = 0 **then**
10: 　　*size_per_warp* ← *element_size* × ワープサイズ
11: 　**else**
12: 　　*size_per_warp* ← *stride* × ワープサイズ
13: 　**end if**
14: 　T ← L2 キャッシュラインサイズ ÷ *size_per_warp*
15: 　**if** $T >$ ワープサイズ **then**
16: 　　T ← ワープサイズ
17: 　**end if**
18: **end for**

図 3.10: トランザクション数の導出

例えば配列 a にアクセスするとき，インデックス式が $e = tid + i$ ならば発生するトランザクション数 T は以下の式で求めることができる：

$$\begin{aligned} T &= \frac{128}{|tid + i - ((tid + 1) + i)| \times \texttt{sizeof(x[}tid+i\texttt{])} \times 32} \\ &= 1 \end{aligned} \tag{1}$$

トランザクション数 Tを基にメモリアクセス命令を以下のように分類する．

- 非コアレッシングアクセス：同一ワープ内の 1 スレッドのみが同一キャッシュライン上のデータにアクセスする．このときトランザクション数は 32 回になる．例を図 3.11 に示す．着色部分は同一ワープ内のスレッドがアクセスする箇所である．
- 不完全コアレッシングアクセス：同一ワープの複数スレッドが同一キャッシュライン上のデータにアクセスする．このときトランザクション数は 32 未満になる．例を図 3.12 に示す．着色部分は同一ワー

プ内のスレッドがアクセスする箇所である．

- 完全コアレッシングアクセス：手順 3 で求めた差の絶対値が 0 のときは不完全ではなく完全なコアレッシングアクセスとする．このとき同一ワープ内の全スレッドが同じアドレスにアクセスし，メモリトランザクション数は 1 になる．例を図 3.13 に示す．着色部分は同一ワープ内のスレッドがアクセスする箇所である．

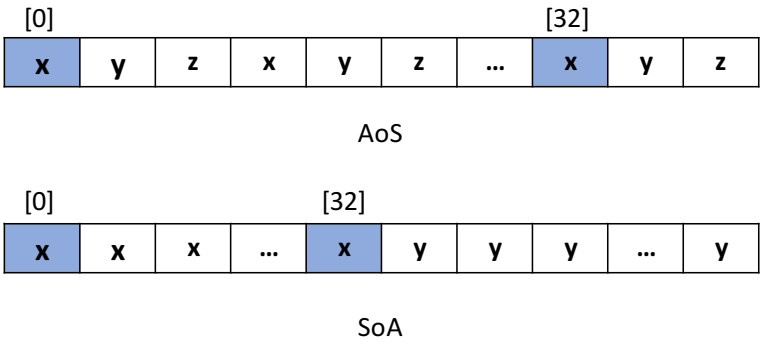

図 3.11: 非コアレッシングアクセスでの AoS と SoA

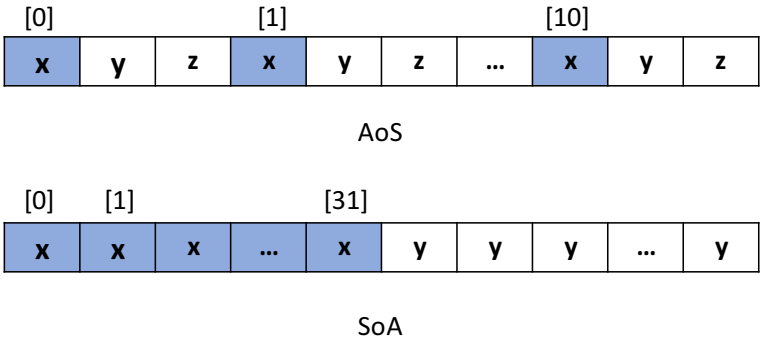

図 3.12: 不完全コアレッシングアクセスでの AoS と SoA

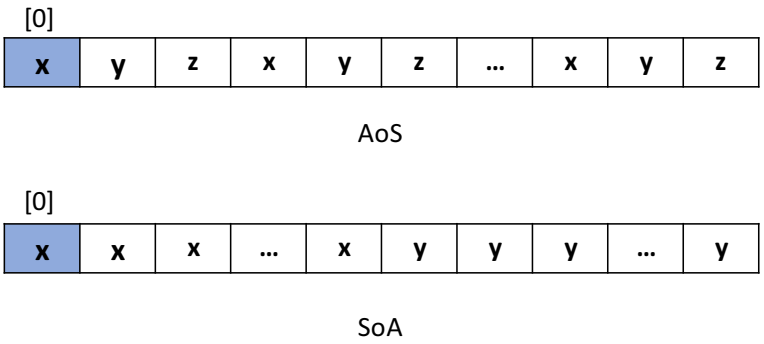

図 3.13: 完全コアレッシングアクセスでの AoS と SoA

3.3 PTX コードの解析

GPU アセンブリ言語である PTX コードを参照することで実際に実行される命令列を取得することができる．この命令列を重み付き制御フローグラフの命令文ノードに割り当てる．その際に，CUDA コード上のループボディと PTX コード上のループボディを対応させる必要があるため，PTX 上のラベルとそのラベルへのジャンプ命令を検出する．さらに PTX 上のメモリアクセス命令と CUDA コード上のメモリアクセス命令を対応させる．これにより PTX 上のメモリアクセス命令のアクセスパターンが分かる．対応付けは PTX 命令のオペランドで指定されるレジスタを参照することで可能である．

プログラムの中にはコンパイラによってループがアンロールされるものがある．この場合，CUDA コードにおけるループボディの実行回数 (N) と PTX におけるループボディの実行回数（N/展開数）は異なる．実行時間の予測では PTX におけるループボディの実行回数が必要である．まず，PTX 上に存在する繰り返し文の条件式の真偽を計算する命令 (setp) を検出する．そして，setp 命令のオペランドにループカウンタ用のレジスタがあるため，その変数に定数を加算する命令 (add) を検出する．この定数が展開数である．

3.4 重み付き制御フローグラフへの命令列の割り当て

図 3.14 の命令文ノードが持つ命令列は，図 3.9 (a) から生成した PTX コードの解析によって取得した命令列の一部を省略したものである．

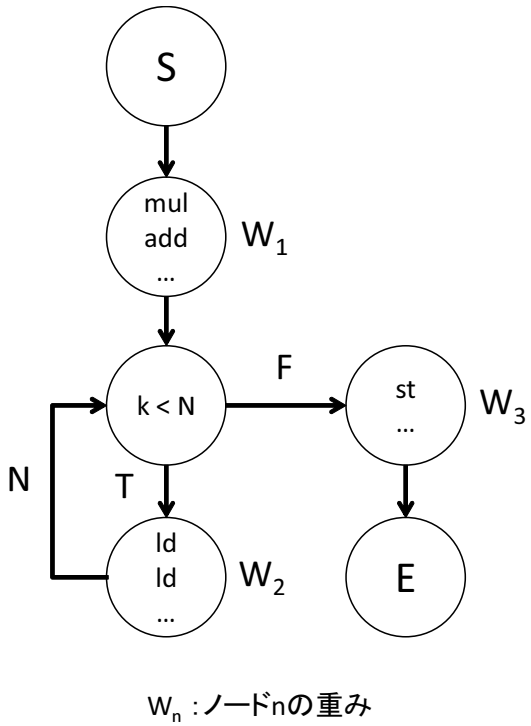

図 3.14: 命令列を割り当てた重み付き制御フローグラフ

次に各命令文ノードの重みを求める．メモリアクセス命令のレイテンシはキャッシュヒット，トランザクション数，命令のオーバーラップによって変化するため，それらを考慮し補正する．これについては以降の 3.5 節で示す．

3.5 命令レイテンシの補正

3.5.1 キャッシュヒット率の導出

キャッシュにヒットした際は DRAM にアクセスする場合と比べてレイテンシが半分以下になるため，キャッシュヒットが期待できるメモリアクセスについてはキャッシュヒット率 (*l2_hit*) を求める．GPU では各ワープが DRAM にアクセスする際，1 回のトランザクションにつきアクセス対象のデータを含む L2 キャッシュラインサイズ (128byte) のデータがキャッシュに格納される．そのため，そのトランザクション以降にそのデータに

アクセスする際，スピルアウトしていない場合はキャッシュヒットとなる．

本手法では簡単化のため L1 キャッシュを考慮しない．また，いずれのループにも含まれないメモリアクセス命令の実行は，プログラムの実行全体で 1 回のみである．そのため性能に与える影響は非常に小さい．また，初回のアクセスはキャッシュミスとなるため，このような命令のヒット率は 0 とする．

初めにネストされていないループにおけるキャッシュヒット率を考える．キャッシュヒット率ごとに配列のインデックス式 e を以下のように分類することができる．

- ループ変数 i_0，...，i_{k-1} のいずれも含まない
  e はループ中で不変となるため最初のアクセス (1 ループ目) 以外はキャッシュヒットとなる．キャッシュヒット率は

$$l2\_hit = \frac{(N-1)}{N} \text{となる．} \tag{2}$$

  - N：命令の実行回数

- ループ変数 i_0，...，i_{k-1} のいずれかを含む
  最初のアクセス (1 ループ目) の際に 128byte のデータがキャッシュに載る．e はループ中で可変であるため，その後のイテレーションにおいて 1 ループ目でキャッシュに載ったデータ以外にアクセスする．その際はキャッシュミスとなるが，同時に 128byte のデータがキャッシュに載る．これを一定周期で繰り返すため $128 \div element\_size$ ループ毎にキャッシュミスが発生し，キャッシュヒット率は

$$l2\_hit = \frac{128 \div element\_size - 1}{32} \tag{3}$$

  となる．

  - $element\_size$：アクセスする配列の *1* 要素当たりのサイズ

次にネストされたループにおけるキャッシュヒットを考える．k 重ループにおける各ループを最も内側のループから外側に向かって 0,...,$k-1$ ループとする．k_1 ループの 1 回目のイテレーションにおいて，0,...,k_1-1 ループでスピルアウトが発生しない場合，2 回目のイテレーションではそれらのデータへのアクセスが全てキャッシュヒットとなる．スピルアウト

```
for(i = 0 ; i < N; i++)
 sum +=a[tid.x].x + a[tid.x].y + a[tid.x].z;
```

図 3.15: AoS アクセスコードの例

が発生した場合は，ネストされていないループと同様に式 (2) もしくは式 (3) によってキャッシュヒット率を求める．

いずれかのループに含まれるメモリアクセス命令のキャッシュヒット率を求めるアルゴリズムを図 3.16 に示す．

本手法ではイテレーション 1 回を終えた時点でスピルアウトが発生したかどうか判定する (図 3.16 7-15 行目)．しかし，ワープスケジューリングによる複雑なワープの挙動は実行しないと分からないため，どのタイミングで発生したかは予想できない．

例えば，x，y，z をメンバとして持つ AoS の全メンバにループ内の各イテレーションでアクセスするコード (図 3.15) を考える．このとき x にアクセスすると y と z もキャッシュに載る．そのため，通常はそれらにアクセスするとキャッシュヒットとなるが，スケジューリング次第では z にアクセスする前に z がスピルアウトしている可能性がある．本手法では 1 回のイテレーションが終わった時点で z がスピルアウトしているかは検出できる．しかし，イテレーション内の実行中にスピルアウトしていても検出できないため本来はキャッシュミスであってもキャッシュヒットとしてしまう．

3.5.2 トランザクション数を考慮したレイテンシの導出

ワープ内での同一の命令に対するトランザクションは図 3.17 のようにオーバーラップされるため，レイテンシは以下の式で求めることができる：

L2 キャッシュにヒットした場合，

$$L_{L2} = l\_l2 + delay\_l2 \times (trans\_num - 1) \tag{4}$$

L2 キャッシュにヒットしない場合，

$$\begin{aligned} L_{DRAM} &= l\_l2 + l\_dram \\ &\quad + delay\_dram \times (trans\_num - 1) \end{aligned} \tag{5}$$

```
1: all ノードの状態 ← 未処理
2: for k_1 = k − 1 to 0 do
3:   for all ノード in k_1 do
4:     if ノードの状態 = 未処理 then
5:       D ← 1 イテレーション当たりのアクセスデータサイズ
6:       for all メモリアクセス命令 in ノード do
7:         if D ≤ L2 キャッシュサイズ then
8:           if i_{k1} と i_{k1} + 1 のときの e が同値 then
9:             キャッシュヒット率 ← 1
10:          else
11:            キャッシュヒット率 ← 式 (2) or 式 (3)
12:          end if
13:        else
14:          キャッシュヒット率 ← 式 (2) or 式 (3)
15:        end if
16:      end for
17:      ノードの状態 ← 処理済
18:    end if
19:  end for
20: end for
```

図 3.16: キャッシュヒット率の導出

- *l_l2* : L2 キャッシュのアクセスレイテンシ
- *delay_l2* : L2 キャッシュへのアクセス命令を実行する際に発生する遅延
- *l_dram* : DRAM のアクセスレイテンシ
- *delay_dram* : DRAM へのアクセス命令を実行する際に発生する遅延
- *trans_num* : トランザクション数

3.5.3 異なる命令のトランザクションのオーバーラップ

異なるメモリアクセス命令であってもそれらに依存関係がない場合はその時点ではストールせず連続で実行される．そのため，オーバーラッ

プされる命令のレイテンシは隠蔽される．3.5.2 項で求めたレイテンシにオーバーラップを考慮すると以下の式のようになる：
L2 キャッシュにヒットする場合，

$$L_{hit} = L_{L2} + delay\_l2 - l\_pre\_l2 \tag{6}$$

L2 キャッシュにヒットしない場合，

$$L_{miss} = L_{DRAM} + delay\_dram - l\_pre\_dram \tag{7}$$

- l_pre_l2：直前に実行されるメモリアクセス命令の l_l2
- l_pre_dram：直前に実行されるメモリアクセス命令の l_dram

よって，あるメモリアクセス命令のレイテンシ L は 3.5.1 項で求めた $l2\_hit$ を用いると以下の式で求めることができる：

$$\begin{aligned} L &= L_{hit} \times l2\_hit \\ &\quad + L_{miss} \times (1 - l2\_hit) \end{aligned} \tag{8}$$

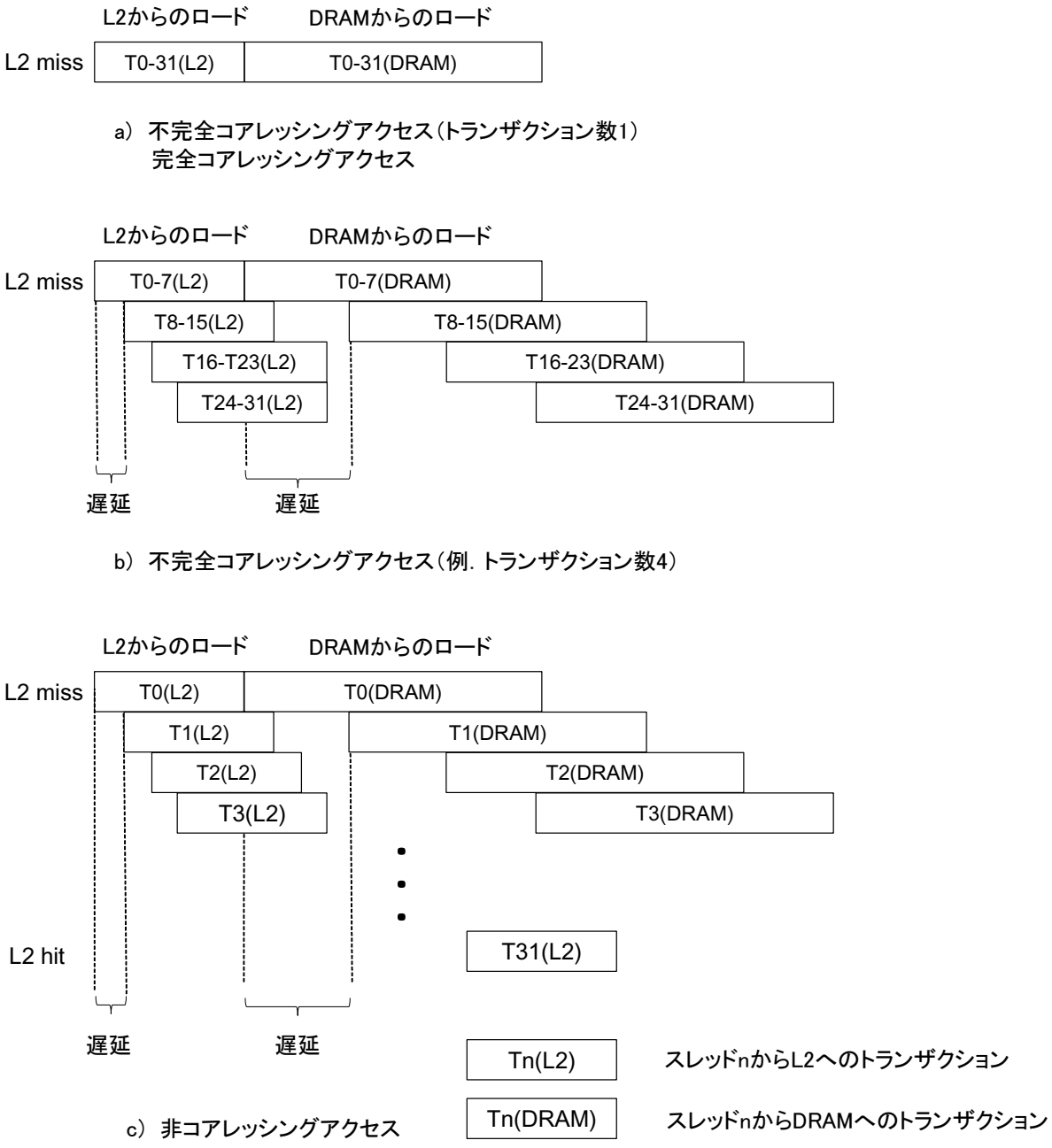

図 3.17: ワープ内でのトランザクションのオーバーラップ

4　評価

提案手法の性能予測の有効性をベンチマークプログラムを用いて実験する．評価環境は評価環境は Intel Xeon CPU E5-1620，メモリ 16GB，GeForce GTX980 [3] を搭載した計算機である．評価ベンチマークを表 4.1 に示す．IDW は逆距離加重を用いる空間補間プログラムである [5]．2MM と 3MM，SYR2K は線形代数プログラム，CORR はデータマイニングプログラムである [4]．これらのプログラムはループ内で正規形を満たすインデックス式 e によって配列にアクセスする．それぞれについてオリジナル版とは別にアライメントを適用した AoS で記述したコードを用意し，実験を行った．

評価結果を表 4.2 に示す．PTX コードの解析を行わない従来手法では IDW，SYR2K，CORR において最適でないレイアウトを選択してしまう．それに対して，提案手法を用いることで候補レイアウトから最適なレイアウトを選択できた．

表 4.1: 評価ベンチマーク

アプリケーション	問題サイズ	概要
IDW	1024^2	逆距離加重空間補間
2MM	8192^2	行列積 (D=A.B; E=C.D)
3MM	8192^2	行列積 (E=A.B; F=C.D; G=E.F)
SYR2K	4096^2	対称行列の階数 2k 更新
CORR	8192^2	相関係数の導出

4.1　スピルアウトによる予測誤差

2MM と 3MM については予測での AoS 比が約 0.92 程度であることに対して，実測では約 0.58 となっている．これは問題サイズの増加によってスピルアウトが発生し，キャッシュミスが起きたためだと考えられる．

配列のインデックス式 e についてイテレーション中のスピルアウトによる影響を分類する．

1. ループ変数 i_0，...，i_{k-1} のいずれも含まない：e はループ中に不変となる．AoS では, あるデータにアクセスした際に同一構造体内の

表 4.2: 評価結果

レイアウト	予測実行時間 (ワープ当たりの cycle)		実測実行時間 (s)
	提案手法	従来手法	
IDW			
AoS	435701791	694375246	14.352
SoA	638328387	638328387	19.106
AoS 比	1.465	0.919	1.331
2MM			
AoS	2669402	2551555	22.449
SoA	2453251	2453251	13.289
AoS 比	0.919	0.961	0.583
3MM			
AoS	16014612	15525002	35.069
SoA	14717582	14717582	19.944
AoS 比	0.919	0.947	0.569
SYR2K			
AoS	1620939	1771407	12.811
SoA	1756687	1756687	14.420
AoS 比	1.084	0.991	1.126
CORR			
AoS	5676632021	15646131301	27.354
SoA	14528418038	14528418038	47.539
AoS 比	2.559	0.928	1.738

他メンバもキャッシュに格納されるため，それらにアクセスする際にキャッシュヒットを期待できる．しかし，それらのデータがスピルアウトした場合に DRAM へのアクセスとなりレイテンシが増加する．

2. ループ変数 i_0，...，i_{k-1} のいずれかを含む：e はループ中に可変となる．ループ変数はインクリメントもしくはデクリメントされるため，レイアウトに関係なく i_{k1} でアクセスした際にキャッシュに格納されたデータを $i_{k1} \pm 1$ でアクセスするとキャッシュヒットとなる．しかし，それがスピルアウトした場合はキャッシュミスとなる．これはレイアウトによらない．

2MM と 3MM は，分類 1 によるアクセスの占める割合が過半数であり，他のベンチマークと比べて AoS でのスピルアウトによる性能低下が大きくなったと考えられる．

5　関連研究

GPU プログラムを対象として，メモリ上のデータレイアウトを自動最適化する研究がある．

Kofler ら [8] は，OpenCL(Open Computing Lan-guage) [16] で記述された GPU コードのデータレイアウトを自動最適化するために Kernel Data Layout Graph (KDLG) を定義し，それを用いた手法を提案している．この手法ではメモリアクセスの局所性を表す KDLG を生成するために，静的解析により必要な情報を取得する．そして，デバイスの L1 キャッシュサイズを基に KDLG を用いた構造体メンバのクラスタリングとレイアウトの決定を行い，GPU コードの自動変換を行う．しかし，動的解析を行わないため，特定のプログラムに対して最適なレイアウトを決定できない可能性がある．例えば，間接参照を用いるものや，実行時の条件分岐によって配列の添え字式の値が変わるプログラムは実行するまでメモリアクセスパターンが不明である．

Weber ら [9] は静的解析と経験的解析のいずれかを使用して GPU コードを最適化する MATOG フレームワークを開発した．MATOG は AoS，SoA，AoSoA をサポートし，最適なレイアウトを選択する決定木を構築する．

それに対して，Fauzia ら [10] は動的解析を用いたメモリアクセス最適化フレームワークを開発した．解析によって各メモリアクセス命令のアクセス先のアドレスを取得し，アドレスが連続していればコアレッシング，そうでなければ非コアレッシングという特徴付けを行った．そして，非コアレシング命令がアクセスする配列の添え字式を書き換えることで，コアレシングアクセスの効果を向上させた．しかし，データレイアウトの変更は実装していない．

上記のデータレイアウト最適化の研究では，AoS，SoA，AoSoA の中からレイアウト選択している．しかし，これら以外にも有用なレイアウトは存在する．Mei ら [5] は，空間補間手法の 1 つである IDW 補間プログラムの高速化として，AoS，SoA，AoSoA，アライメントされた AoS(AoS-align) などでの評価を行った．この中でも，naive において AoS-align は AoS，SoA の両方と比べて高い性能を発揮した．我々の研究では SoA と AoS-align の中から最適なレイアウトを決定する．

GPU プログラムの性能予測については多くの研究努力がなされている [11–13]．Hong ら [6] は CUDA コードから実行パフォーマンスを予測するための GPU アーキテクチャの解析モデルを提案した．このモデルは

ワープスケジューリングによるレイテンシの隠蔽をモデル化している．しかし，メモリ命令のアクセスレイテンシが変化しない初期の GPU アーキテクチャ向けに提案されている．現在の GPU は複数の階層のキャッシュを持つため，アクセスする際のアクセス先によってレイテンシは変化する．従って，このモデルは現在の GPU アーキテクチャに適用できない．

Wang ら [7] は C で書かれた関数の GPU 上での動作を推定し，性能を予測するフレームワークである CGPredict を提案している．このフレームワークでは入力である C コードに対して動的解析を行い，実行時の情報をトレースする．そのトレース結果を基に GPU 上で並列実行した際の動作を推定する．一般に二重ループを並列実行する場合は内側ループにスレッドが割り当てられ，その部分が並列実行となる．このとき，CUDA でのワープは C コードでの内側ループの連続した 32 インデックス単位にまとめたものと解釈できる．例えばインデックスが 0 から 127 までのループを並列化するとき，スレッドインデックスは 0 から 127 となり，合計ワープ数は 4 となる．こうすることで逐次実行である C コードの実行時情報から並列実行である CUDA の動作を推定できる．しかし，このフレームワークは C での実行時情報を使用するため，CUDA コンパイラによる最適化を考慮できない．また，実行時間の数百倍の解析時間が必要となるため，複数の入力ファイルを想定としたプログラムに適さない．

これらの研究では CUDA コードの静的解析もしくは動的解析を用いるが，CUDA コンパイラによる最適化を考慮していない．例えばアライメントによって複数の命令がまとめられるケースでは実際に実行される命令数と解析によって予測された命令数に誤差が発生する．そのため，これらの研究で用いられる手法をレイアウト選択のために使用すると最適でないレイアウトを選択する可能性がある．本研究では GPU アセンブリコードである PTX コードの解析を静的解析と併用することでコンパイラによる最適化に対応している．

6　まとめと今後の課題

本研究では GPU プログラムのデータレイアウト最適化のための性能予測解析手法を提案し，評価を行った．その結果，従来手法では予測することができないアライメントによるアクセス最適化性能を予測することができた．また，高精度で最適なレイアウトを選択することが可能となった．

今後の課題として，if 文のような分岐への対応が挙げられる．分岐によってレイアウト最適化対象の構造体メンバへのアクセス回数が変わる場合はそれを考慮し，実行時間を予測する必要がある．このとき，各分岐先を実行する確率を予測することができれば，それを重み付き制御フローグラフのノードが持つ重みの補正に用いることで対応できる．そのために静的解析と動的解析を併用する必要がある．また，今回評価に用いたベンチマークではキャッシュにおけるスピルアウトによる予測誤差が原因となり最適でないレイアウトを選択することはなかった．しかし，いくつかのベンチマークでは誤差が生じているためより多くのアプリケーションでの評価とスピルアウトのモデル化が必要である．

謝辞

本研究を行うに辺り, 御指導, 御助言頂きました大野和彦講師, 並びに多くの助言を頂きました山田俊行講師に深く感謝致します. また, 様々な局面にてお世話になりました研究室の皆様にも心より感謝致します.

参考文献

[1] *GPGPU.org: General-Purpose computation on Graphics Processing Units*. http://www.gpgpu.org/, (2018.2.6).

[2] *NVIDIA Developer CUDA Zone*. https://developer.nvidia.com/cuda-zone, (2018.2.6).

[3] *Whitepaper NVIDIA GeForce GTX980*. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, (2018.1.9).

[4] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos, *Auto-tuning a high-level language targeted to GPU codes*. In 2012 Innovative Parallel Computing (InPar ' 12).IEEE,1-10,(2012).

[5] Mei, Gang, and Hong Tian, *Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation*. SpringerPlus 5.1,104,2016.

[6] Sunpyo Hong and Hyesoon Kim, *An Analytical Model for a GPU Architecture with Memory-level and Threadlevel Parallelism Awareness*. In Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA ' 09). ACM,152163,2009.

[7] Wang, Siqi, Guanwen Zhong, and Tulika Mitra, *CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications*. ACM Transactions on Embedded Computing Systems (TECS),16.5s,146,2017.

[8] Kofler, Klaus, Biagio Cosenza, Thomas Fahringer, *Automatic data layout optimizations for gpus*. European Conference on Parallel Processing. Springer,263-274,2015.

[9] Weber, Nicolas, Sandra C. Amend, Michael Goesele, *Guided profiling for auto-tuning array layouts on GPUs*. Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems. ACM,9,2015.

[10] Fauzia, Naznin, Louis-Nol Pouchet, P. Sadayappan, *Characterizing and enhancing global memory data coalescing on GPUs*. Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society,12-22,2015.

[11] Baghsorkhi, Sara S., et al, *An adaptive performance modeling tool for GPU architectures*. ACM Sigplan Notices. Vol. 45. No. 5. ACM,105-114,2010.

[12] Arun Kumar Parakh, M Balakrishnan, and Kolin Paul, *Performance Estimation of GPUs with Cache*. In 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum,23842393,2012.

[13] Gene Wu, Joseph L Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou, *GPGPU performance and power estimation using machine learning*. In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA ' 15). IEEE, 564576,2015.

[14] Sung, I-Jui, Geng Daniel Liu, and Wen-Mei W. Hwu, *DL: A data layout transformation system for heterogeneous computing*. Innovative Parallel Computing (InPar), 2012. IEEE, 1-11,2012.

[15] Michael Andersch, Jan Lucas,Mauricio Alvarez-Mesa, Ben Juurlink, *Analyzing GPGPU Pipeline Latency*. http://lpgpu.org/wp/wp-content/uploads/2013/05/poster_andresch_acaces2014.pdf, (2018.2.6).

[16] *OpenCL Overview*. https://www.khronos.org/opencl/, (2018.2.6).