Vol.2019-ARC-236 No.2
2019/6/11

メモリアクセスパターンを考慮した GPUスケジューリングポリシーの選択手法

川口 優樹[1]　津邑 公暁[1]

概要：GPU は多数の計算ユニットを並列に動作させることによってパイプラインやメモリアクセスレイテンシを隠蔽し，CPU と比較して高いスループットを実現するが，これには計算ユニットへ命令を割り当てるスケジューリングポリシーが重要である．近年様々なスケジューリングポリシーが研究されているが，既存の GPU スケジューラは Warp の実行順序やメモリアクセスパターンなどの各プログラムが持つ様々な特徴を考慮せずにすべてのプログラムに対して同じスケジューリングポリシーを適用するため，GPU の計算ユニットの利用効率が必ずしも高くない場合がある．そこで本論文では，アセンブリコードに対してメモリアクセスを解析し，その結果に基づき，プログラムごとにスケジューリングポリシーを選択する手法を提案する．提案手法を用いて評価を行った結果，既存の GPU スケジューラと比較して 34.2% の実行時間削減を達成することを確認した．

1. はじめに

GPU（Graphics Processing Unit）は画像処理に特化したプロセッサである．一般的に，画像処理は画素等の膨大な数の処理対象に対して定式化された単純な演算を繰り返すことが多い．このため，GPU は簡素な設計の計算ユニットを多数搭載し，多数の処理対象に対して同時に命令を適用することで，画像処理の高速化を図っている．そのため，GPU の計算ユニットは，今日の CPU に標準搭載されているプリフェッチや投機実行，レジスタ・リネーミング等の機能を備えていない．このように，GPU は並列処理に特化したアーキテクチャであるため，データ並列性の高い演算処理を実行する場合，複雑な構成のプロセッサである CPU と比べて，少ない消費電力で高いスループットを実現できる．また，GPU は多数の計算ユニットを並列に動作させることによって，メモリアクセスレイテンシを隠蔽する．計算ユニットへ命令を割り当てるスケジューリングによってメモリアクセスタイミングや命令フェッチのタイミングが大きく変化するため，メモリアクセスレイテンシを隠蔽し，GPU が持つ性能を引き出すには，このスケジューリングが重要である．そのため，近年，様々なスケジューリングポリシーが研究されており，中でも **LRR**（**Loose Round Robin**）や **GTO**（**Greedy Then Oldest**）が一般的に知られている．しかし，NVIDIA 社 [1] 製の GPU に搭載されている **Warp Shceduler** と呼ばれるハードウェアスケジューリングユニットは，メモリアクセスパターンや依存関係などの各プログラムが持つ特徴を考慮せずに．すべてのプログラムに対して同じスケジューリングポリシーを適用するため，パイプラインやメモリアクセスレイテンシなどが必ずしも隠蔽できているわけではなく，GPU の計算ユニットの利用効率が必ずしも高くない場合がある．

本研究では，一般的に広く用いられている，NVIDIA 社が開発した並列計算アーキテクチャモデル **CUDA**（**Compute Unified Device Architecture**）[2] をターゲットとし，プログラマが記述したプログラムを静的解析した結果に基づきプログラムごとにスケジューリングポリシーを変更することで GPU のスループット向上を図る．

2. GPU の概要

本章では，まず CUDA を採用した GPU の構造および，CUDA の実行モデルについて述べる．

2.1 GPU の内部構成

GPU は多数の演算器を並行して動作させることで高い演算性能を実現する．本節では，CUDA をサポートしている NVIDIA の GPU 内部に存在する計算ユニットの基本的な構成を示す．**図 1** に示すとおり，NVIDIA の GPU は階層的なプロセッサ構成を採用しており，GPU 内部には，共有 L2 キャッシュと多数の **SM**（**Streaming Multi-Processor**）と呼ばれる計算ユニットが存在する．SM は

[1] 名古屋工業大学
Nagoya Institute of Technology

© 2019 Information Processing Society of Japan

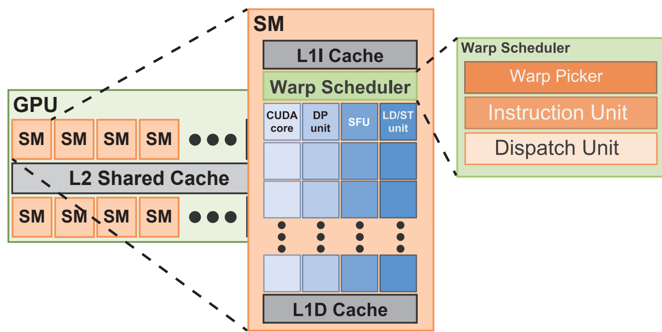

図 1　GPU の構成図

多数の実行ユニットとキャッシュおよび，各実行ユニットへ命令割当を行う **Warp Scheduler** から構成される．

Warp Scheduler は，キャッシュから Warp を取得する Warp Picker，Warp Picker が取得した Warp を 1 命令単位に分解し，命令解釈を行う Instruction Unit および，Warp を実行ユニットに割り当てる Dispatch Unit から構成される．実行ユニットには，演算処理ユニットである CUDA Core や，倍精度演算命令を実行する DP Unit，超越関数演算などの複雑な命令を実行する SFU（Special Function Unit），キャッシュメモリに対するロード/ストア命令を実行する Load/Store Unit が存在する．SM 内部に存在するこれらの実行ユニットには，Register と呼ばれる高速かつ小容量な記憶回路がそれぞれユニット毎に独立して併設されている．この実行ユニットと Register とを合わせて Lane と呼ぶ．実行ユニットのうち SM 内に最も多く搭載されている CUDA Core は，32bit 単精度浮動小数点の積和演算や，32bit 整数の加減算と論理演算を実行するユニットであり，CPU のコアに比べて非常に簡素な設計になっている．このことが GPU の省電力性に寄与している．

2.2　CUDA 実行モデル

CUDA のプログラミングモデルでは，GPU のプロセッサの各階層に対して処理単位を割り当てる．この処理単位のなかで最小の単位を Thread と呼び，Thread の集合を Block，Block の集合を Grid と呼び，Grid は GPU に，Block は SM に，Thread は Lane に割り当てられる．CUDA では階層的に Thread 群を管理することで，プログラマが GPU の複雑なプロセッサ構造を意識することなく GPU を用いることを可能にしている．この CUDA の開発環境を用いて，プログラマが GPU プログラムを開発する際には，CPU に割り当てる処理と GPU に割り当てる処理とを分けて記述する必要がある．CUDA では CPU をホスト，GPU をデバイスと定義しており，CPU 上で実行されるコードはホストコード，GPU 上で実行されるコードはデバイスコードと呼ばれる．このデバイスコードでは，プログラマが GPU 上の各 Thread にそれぞれ割り当てる処理単位を関数として定義する．CUDA では，この関数のことを**カーネル関数**と呼び，カーネル関数はホストコード側から呼び出される．CUDA では 1 つのカーネル関数を大量の Thread に割り当て，同じ命令を別々のデータに対して実行する．

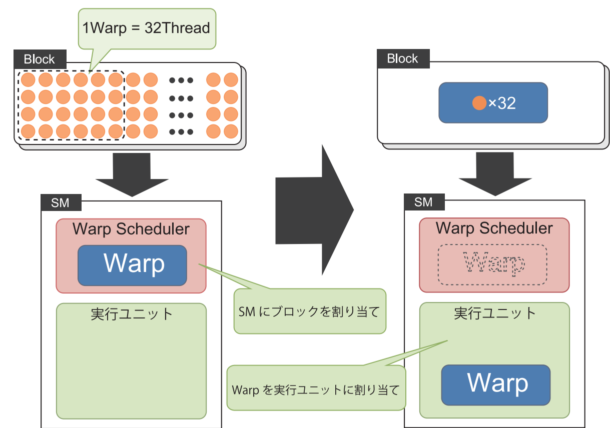

図 2　Warp の割り当て過程

CUDA では同時に同じ命令を実行する Thread 群の 1 単位を **Warp** と呼び，1Warp は 32 本の Thread からなる．Warp にはロード命令や算術命令など様々な命令がまとめられており，それらの命令を実行する際に要するサイクル数は異なるため，Warp ごとに実行サイクル数も異なる．この Warp は Warp Scheduler によって，SM 上の Lane に割り当てられ，カーネル関数を実行する．ここで，CPU が CUDA プログラムの実行を開始してから，Warp 内の各 Thread がカーネル関数を実行するまでの流れを説明する．まず CPU がカーネル関数を呼び出すと，GPU 上に複数の Block が生成され，各 Block 内に Thread 群が生成される．そして，SM に Block が割り当てられると，Block 内の Thread は Warp Scheduler によって Warp 単位にまとめられる．さらに Warp Scheduler は各 Warp が実行する予定のカーネル関数を 1 命令単位に分解し，それぞれの命令が実行可能に成り次第，SM 上の複数の Lane に Warp を割り当てる．そして，Warp は Lane 群内の実行ユニットや Register を用いて命令を実行する．この際，Warp Scheduler は Lane を 16 あるいは 32 個まとめた Lane Set に対して，Warp を割り当てる．しかし，Warp Scheduler がある Warp を Lane Set に割り当てようとする際，演算で利用するデータのロードが完了していない等の理由により，Warp が命令を実行することができず待機状態となる場合が存在する．このような場合，Warp Scheduler は待機状態となった Warp の代わりに別の Warp を Lane Set に割り当てる．ただし，命令を実行可能な Warp が存在しない場合は，Warp Scheduler は Warp を Lane Set に割り当てることができない．このような場合，Warp の命令実行に用いられていない Lane Set があれば，その Lane Set は遊休（idle）状態になってしまう．したがって GPU のスループットを向上することにとって，Warp を割り当てる順序や切り替えのタイミングなどが重要である．なお，Warp を割り当てる順序や切

© 2019 Information Processing Society of Japan

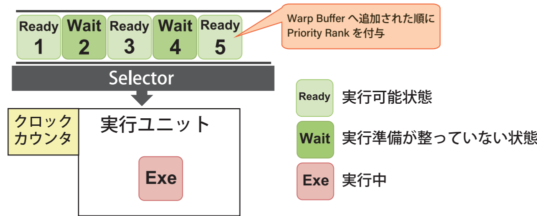

図 **3**　LRR を実装した Warp Scheduler の概略図

り替えのタイミングのことを**スケジューリングポリシー**と呼び，**LRR（Loose Round Robin）**，**GTO（Greedy Then Oldest）**が一般的に知られている．

3.　スケジューリングポリシーの概要

本章では，LRR，GTO それぞれのスケジューリングポリシーについて，それぞれの具体的な動作について述べた後，シミュレータによるスループットの計測結果に基づき，スケジューリングポリシーとスループットとの関係について議論する．

3.1　Loose Round Robin

LRR はタイムクォンタムと呼ばれる一定の時間で実行ユニットから Warp を追い出し，Buffer 内の Warp をラウンドロビン法に従って順繰りに実行するスケジューリングポリシーである．ここで，LRR を実装した Warp Scheduler の概略図を**図 3** に示す．なお，Warp が待機する Buffer は，命令キャッシュであり，実際には Warp Scheduler 外に位置するが，説明の都合上，Warp Scheduler の中にあるものとし，以降 **Warp Buffer** と呼称する．また，2.1 節で述べた Warp Picker および Instruction Unit，Dispatch Unit は説明の都合上，それら全ての機能を備えた単体のユニットとして集約し，**Selector** と呼称する．LRR を実装した Warp Scheduler はタイムクォンタムの終了を検出するクロックカウンタおよび，Warp Buffer 内から次に実行する Warp を選択する Selector から構成される．また，Warp Buffer 内には実行可能状態である Warp と，依存関係などにより実行準備が整っていない Warp が存在する．また，各 Warp には Warp Buffer へ追加された順に **Priority Rank** が付与される．Priority Rank とは，Selector が Warp Buffer 内から Warp を選択する際に使用する指標であり，Selector は Warp Buffer 内から Priority Rank が最も低い実行可能状態である Warp を実行ユニットへ割り当てる．Rank 値は優先順位を表していて，その数値が小さいものから割り当てる．

ここで，**図 4** を用いて LRR の Warp 割り当て順序について述べる．

図 4 は Warp 切り替え時における Warp Buffer 内の様子を表している．なお，初期状態（t_0）では，$Warp_3$ が実行中であり，Warp Buffer 内では図 4 中のように Warp が待機しているとし，簡単化のために Priority Rank は 1 から順に付与している．まず，$Warp_3$ がタイムクォンタムを終了し，実行ユニットから追い出されたとする．このとき，$Warp_3$ は Priority Rank として 5 を付与され (t_1)，Warp Buffer へと送られる．Selector は Warp Buffer 内の最も優先度が高い Warp である $Warp_4$ を選び，実行ユニットへ割り当てる (t_2)．実行が進み，$Warp_4$ がタイムクォンタムを終了し，Priority Rank として 9 を付与されて，Warp Buffer 内に追加されたとする (t_3)．このとき，Selector は次に実行する Warp として最も優先順位が高い $Warp_1$ を選択するが，$Warp_1$ は実行準備が整っていないため (t_4)，$Warp_1$ の Priority Rank を 7 に更新し，割り当てをスキップする．その後，Selector は $Warp_1$ の次に優先度が高く実行可能状態である $Warp_2$ を実行ユニットに割り当てる (t_5)．

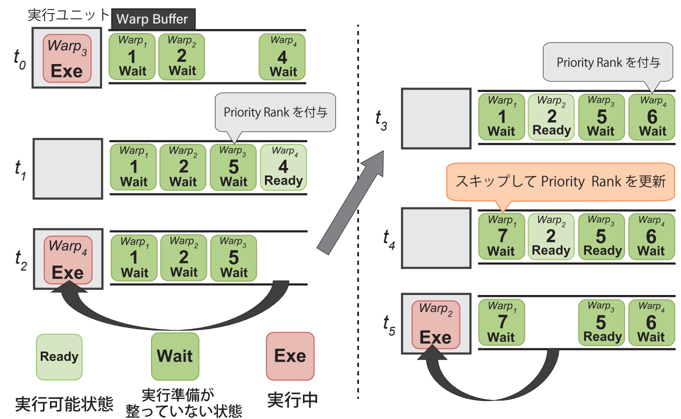

図 **4**　LRR における Warp Buffer の遷移

warp 感で同じデータにアクセスするような場合，短いタイムクォンタムで Warp を順に切り替えながら実行する LRR では，切り替え前の Warp が使用していたキャッシュ上のデータを切り替え後の Warp が利用することができるため，キャッシュヒット率が高くなる．ただし，Warp の実行時間がタイムクォンタムより短い場合，タイムクォンタムの長さは固定であるため，Warp の実行終了からタイムクォンタムの終端までの期間，実行ユニットがアイドル状態となり，実行ユニットの利用効率が低くなる．

3.2　Greedy Then Oldest

GTO は依存関係やメモリアクセスのために Warp が Stall した際に，実行ユニットからその Warp を追い出し，Warp Buffer へ最も早く追加された Warp を実行するスケジューリングポリシーである．なお，LRR と同様に説明の都合上 Warp Buffer は Warp Scheduler 内に位置するものとする．GTO を実装した Warp Scheduler は，実行ユニット内で Warp が Stall したことを検出する Stall 検出器および，Warp Buffer 内から次に実行する Warp を選択する Selector から構成される．Warp Buffer 内には LRR

© 2019 Information Processing Society of Japan

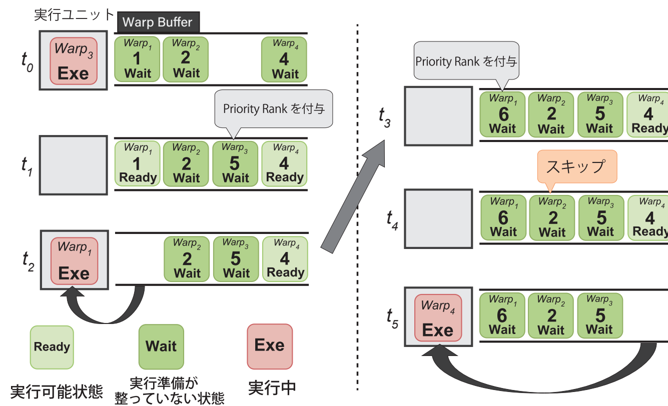

図 5　GTO における Warp Buffer の遷移

表 1　シミュレータ諸元

Processor	GTX480
SM	30 units
clock	700 MHz
Warp Scheduler	2 units
D1 cache	48 KBytes
ways	4 ways
D2 cache	8 MBytes
ways	8 ways
clock	700 MBytes
Memory	1536 MBytes
clock	924 MHz
latency	100 cycles
Interconnect network clock	700 MHz

と同様に，実行可能である Warp と実行準備とが整っていない Warp が存在し，各 Warp には Warp Buffer へ追加された順に Priority Rank が付与されている．Selector は Warp Buffer 内から Priority Rank 値が最も小さい Warp を実行ユニットへ割り当てる．**図 5** は Warp 切替時における Warp Buffer 内の状態を表している．なお，初期状態（t_0）では，$Warp_1$ が実行中であるとし，Warp Buffer 内の Warp には 1 から順に Priority Rank を付与している．まず，$Warp_1$ がメモリアクセス等を原因として Stall したとする．このとき，$Warp_1$ は Warp Buffer 内へ送られ，Priority Rank として 5 が割り当てられる (t_1)．その際，Selector は Warp Buffer 内の最も優先度が高い Warp を選び，実行ユニットへ割り当てる．この例では，$Warp_1$ が最も優先度が高い Warp であるため，Selector は $Warp_1$ を実行ユニットへ割り当てる (t_2)．実行が進み，$Warp_1$ が実行ユニットから追い出され，Warp Buffer に戻されたとする (t_3)．この際，Warp Buffer 内で最も優先順位の高い Warp は $Warp_2$ であるが，実行可能状態出ないためスキップされ（t_4），次に優先順位が高く実行可能状態にある $Warp_4$ を選択肢，実行ユニットに割り当てる（t_5）．LRR ではこの際に実行を Selector が実行をスキップした Warp に対して Priority Rank を更新するが，GTO では更新しない．

このように Warp Buffer 内に追加された順に付与される Priority Rank を更新せずに用いることにより，GTO は Warp Buffer 内に最も早く追加された Warp を常に優先的に実行する．

GTO の特徴として，LRR よりも Warp の切り替え総数が少なくなるため，切り替えに要するオーバーヘッドが全体性能に与える影響は小さく抑えられる．しかし Warp ごとの進行度に大きな差が生じやすいため，切り替え前の Warp と切り替え後の Warp とで処理対象データが異なる可能性が高くなり，キャッシュミス率が LRR よりも高くなる傾向にある．

3.3　スケジューリングポリシーとスループットとの関係

スケジューリングポリシーとスループットとの関係を調査するために，実行時間と 1000 命令あたりの L2 キャッシュミス回数である MPKI（Miss Per Kilo Instraction）を計測する．

3.3.1　評価環境

評価環境を**表 1** に示す．本評価では GPU シミュレーション環境として広く用いられている GPGPU-Sim[3] を拡張した，Mafia シミュレータ [4] を用いて評価した．想定する GPU は Fermi アーキテクチャ [5] を採用した GTX480 とし，OS は Ubuntu16.04 とした．また，ベンチマークプログラムには，CUDA SDK [6] から SCP（Scalar Product），GUPS（Random Access）[4]．Parboil ベンチマーク [7] から HISTO（2D Histogram），SAD（Sum of Absolute Difference）を使用した．なお，コンパイラにおける最適化オプションはすべて-O3 を指定した．

3.3.2　評価と考察

各ベンチマークプログラムをスケジューリングポリシーを変更して計測した評価結果を**図 6** に示す．図 6 はそれぞれ各ベンチマークプログラムにおける MPKI，実行時間を 2 本のグラフで表しており，これらのグラフはそれぞれ，

（GTO） スケジューリングポリシーを Greedy Then Oldest として実行

（LRR） スケジューリングポリシーを Loose Round Robin として実行

した際の結果を示している．なお，評価結果は近年の GPU で広く用いられている GTO の結果を 1 として正規化している．

図 6 中 (a)，(b) より，HISTO および SAD に関しては **（GTO）** と比較して **（LRR）** のキャッシュミス回数が少なく，実行時間も短いことが確認できる．3.1 節で述べた LRR の特徴より，これらのベンチマークでは，Warp 間のデータ局所性が存在し，キャッシュが有効に働く．

一方で，SCP および GUPS に関しては，（LRR）のキャッ

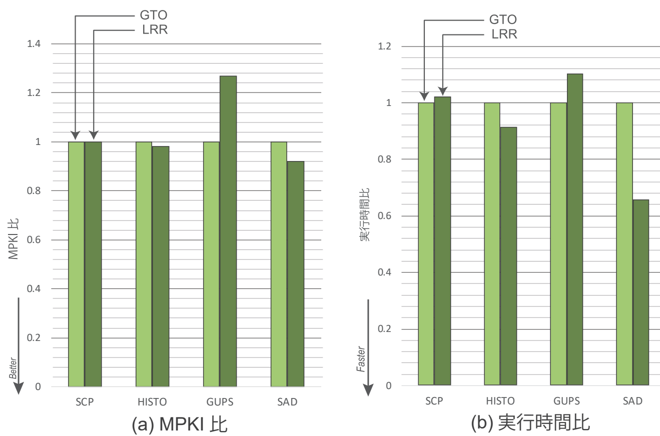

図 6 (a)MPKI 比 (b) 実行時間比

シュミス回数が（GTO）と比較して同じかそれ以上であり，実行時間も（LRR）のほうが長くなっている．3.1 節で述べた LRR の特徴より，これらのベンチマークでは，キャッシュが有効に働かないため，LRR の特徴が生かし切れず，GTO と比較してスループットが低下していると考えられる．以上のことから，プログラムごとに有利なスケジューリングポリシーは異なることが分かる．

そこで本論文では，プログラムをメモリアクセスに着目して事前に静的解析を行い，得られたメモリアクセスに関する特徴から，プログラムを実行する上で LRR，GTO どちらのスケジューリングポリシー適しているか判断する．判断結果に基づき，GPU で実際に実行する際にプログラムごとにスケジューリングポリシーを変更することで GPU のスループットを向上する手法を提案する．

4. 静的解析に基づく GPU スケジューリングポリシーの選択

本章では静的解析に基づいたスケジューリングポリシー選択手法の提案と，解析に使用する擬似アセンブリ言語の特徴について述べる．

4.1 概要

3.3 節で述べたように，Warp 間で同一データに対するアクセスが多いプログラム，つまりデータ局所性が大きいプログラムである場合，キャッシュが有効に働く点から LRR のほうが有利である．一方，データ局所性が小さいプログラムである場合，キャッシュが有効に働かず，LRR の特徴を生かし切れないため，GTO が有利であると考えられる．そこで本論文では，プログラムを静的解析しデータ局所性の大小を見積もることによって実行前に適切なスケジューリングポリシーを選択する手法を提案する．なお，本システムは静的解析部とポリシー切替部から成る．

まず，静的解析部では，プログラマが記述したコードを静的解析してデータ局所性の大小を見積もる．なお CUDA C はその抽象度が高いため，今回目的とする，アクセス対象となるメモリアドレスの解析を行うには不向きである．そこで，CDUA C コンパイラが出力する **PTX(Parallel Thread Execution)**[8] と呼ばれる疑似アセンブリコードを解析する．PTX では，GPU の動作を命令レベルで把握しやすい上，ロードやストアなどのメモリアクセスが発行されるタイミングやその対象アドレスを把握することが可能である．

なお，以降では PTX で記述されたアセンブリコードを **PTX コード**と呼称する．解析後，得られた結果から GTO，LRR のどちらのスケジューリングポリシーが適しているかを判断し，判断した結果を PTX コードに専用命令として埋め込む．

次にポリシー切替部では，静的解析で埋め込んだ命令に基づいて Warp Scheduler のスケジューリングポリシーをプログラムごとに変更する．そのためコンパイラが出力した専用命令によってスケジューリングポリシーが変更できるよう GPU 上の Warp Scheduler を拡張する．

4.2 静的解析部

Warp 間におけるデータ局所性の大小を見積もるにあたり，コード中に現れる各アドレスに対して同一アドレスに対する平均アクセス回数を概算することとした．具体的には，PTX コード内のロードストア命令に着目して，コード内で発生する同一アドレスに対するアクセス回数をそれぞれカウントする．ただし，静的解析では，ループや条件分岐によるメモリアクセス回数の動的な変動を追跡しきれない．そこで本手法では，ループや条件分岐内で発生するメモリアクセスについては，そのネストの深さに応じた重み付けをしてアクセス回数を概算する．求めた平均アクセス回数が経験則に基づいて定めたしきい値未満であれば GTO，以上であれば LRR が有利であると判断し，その結果を専用命令として PTX コードに埋め込む．

キャッシュは，一度アクセスしたアドレスの近傍をまとめてバッファするため，同一アドレスに対するアクセスではない場合でも，一度アクセスしたアドレスの近傍であれば，キャッシュにデータが残っている可能性が高い．PTX コード内のメモリアドレスは，レジスタに格納されたベースアドレスに対してオフセットを加減算することにより，実際にアクセスするアドレスを求める，相対アドレッシングによって記述されている．そのため，本手法では，レジスタに格納されたベースアドレスを追跡し，同一ベースアドレスを用いたメモリアクセスを同一キャッシュブロックに対するメモリアクセスとみなすこととする．例えば**図 7** の PTX コード中では，1 行目から 3 行目まで，$\%rd10$ レジスタを介したメモリアクセスが発生しており，1 行目から 3 行目までで $\%rd10$ レジスタ内の値は変更されていないため，これらのアクセスはすべて同一ベースアドレスを用いたメモリアクセスである．よって，1 行目から 3 行目まで

```
1 ld  %r4, [%rd10+0];
2 ld  %r5, [%rd10+4];
3 ld  %r6, [%rd10+8];
4 add %rd10, %rd10, 40;
5 ld  %r7, [%rd10+16];
```

図 7 本手法で同一アドレスとみなすアドレスの例

```
 1 $func1:
 2     mov %rd1, 10;
 3     mov %rd2, 100;
 4     mov %rd3, 1000;
 5     ld  %r2, [%rd1+0];
 6     setp.eq %p1, %r2, %r3;
 7     @%p1     bra $Label1;
 8     ld  %r4, [%rd2+0];
 9 $Label1:
10 $Label2:
11 //<loop>
12     ld  %r7, [%rd3+0];
13     ld  %r4, [%rd2+4];
14     setp.eq %p3, %r7, %r8;
15     @%p3     bra $label2;
16     add %rd3, %rd3, 40;
17     ld  %r7, [%rd3+16];
```

図 8 コード例

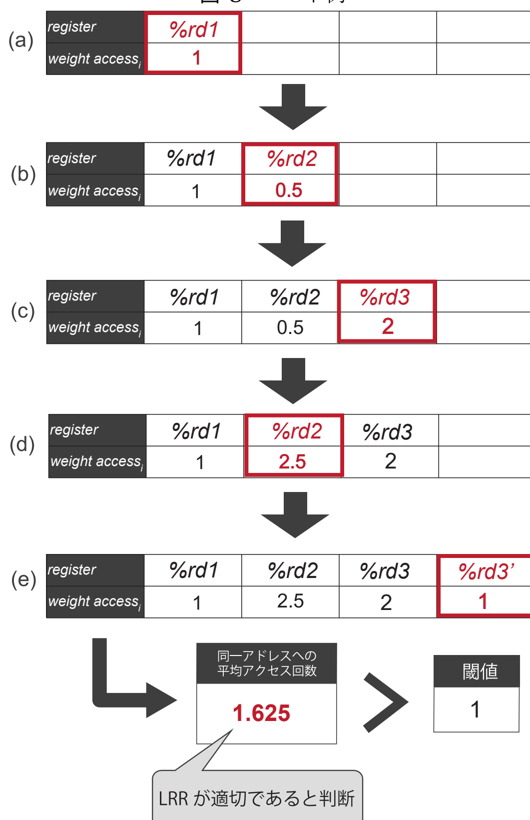

図 9 アクセス先ごとに記憶するアクセス回数の遷移

のメモリアクセスは，すべて同一キャッシュブロックに対するメモリアクセスとみなす．一方，4 行目では $\%rd10$ レジスタ内の値を変更しているため，以降，$\%rd10$ レジスタを介したメモリアクセスは 1 行目から 3 行目までにアクセスしたメモリアドレスとは異なるブロックに対するメモリアクセスと捉える． ループおよび条件分岐のネストの深さ

```
1 .pol    "LRR"; //専用命令
2 $func1:
3     ld  %r2, [%rd1+0];
4     ld  %r3, [%rd1+4];
5 ...
```

図 10 専用命令を埋め込んだ PTX コード

に応じた，アクセス回数の重み付けに関しては，ループネストが深いほど重く，分岐構造ネストが深いほど軽くなるよう，式 1 のように定義した．

$$w_{a,i} = \frac{2^{LDepth_{a,i}}}{2^{BDepth_{a,i}}} \tag{1}$$

ここで $w_{a,i}$ は，ベースアドレス a を用いて参照されるブロック $block_a$ に対する i 回目のアクセスに対して乗じる重みであり，$LDepth_{a,i}$，$BDepth_{a,i}$ はそれぞれ，$block_a$ に対する i 回目のアクセスが発生する際の，ループおよび分岐構造ネストの深さを表す．なお，ループおよび分岐構造外でのアクセスでは，$LDepth$ および $BDepth$ ともに 0，この際の w は $2^0/2^0 = 1$ となる．そして $block_a$ に対するこの $w_{a,i}$ の総和を，$clock_a$ に対する重みつきアクセス解すとし，式 2 のように定義する．

$$wcount_a = \sigma_i w_{a,i} \tag{2}$$

ここで，**図 8** に示す PTX コードを例に，重み付きアクセス回数の算出手順について詳しく説明する．

まずコードの 5 行目において，$\%rd1$ レジスタを介したメモリアクセスが行われるが，この部分のネストの深さは 0 であるため，重みは 1 となる．また，これが $\%rd1$ を介した最初のアクセスであるため，$wcount$ も 1 となる．これを**図 9** 中（a）に示すような表の形で，解析中保持しておく．

解析を進め，7 行目において分岐命令を検出すると，次行にループ開始部を示すコメントが存在しているか否かを確認する．分岐命令は条件分岐とループ開始部の両方で使用されうるが，CUDA C コンパイラが出力する PTX コードでは，ループ開始部にはそれを示すコメントが挿入されるため，この有無でループであるか否かを判定できる．この例において 7 行目直後にコメントは挿入されていないため，条件分岐であると判断し，このあとのアクセスに対しては $BDepth$ の値を 1 として計算する．よって，8 行目で行われる，$\%rd2$ レジスタを介したメモリアクセスに対しては，式 12 に基づき $wcount = 2^0/2^1 = 0.5$ と算出され，これを新たに記憶する（図 9(b)）．

次に 9 行目でラベル Label1 が検出されるが，これは先ほど 7 行目の分岐命令で分岐先として指定したラベルである．よってこれ以降は再び $BDepth$ を 0 として重み計算を行う．

10 行目のラベル Label2 に関しては，次行にループ開始部を示すコメントが挿入されているため，次行以降をルー

© 2019 Information Processing Society of Japan

プ内の処理とみなし，これ以降は $LDepth$ を 1 として重み計算を行う．よって 12 行目の %$rd3$ を介したメモリアクセスに対しては，$w = 2^1/2^0 = 2$ と計算され，これが %$rd3$ を介した最初のアクセスであるため，この時点での $wcount$ も 2 となる（図 9(c)）．

次に 13 行目において，再び %$rd2$ を介したメモリアクセスが行われる．このアクセスに対する w も 2 である為，%$rd2$ を介したアクセスに対してすでに記憶されているここまでの $wcount$ の値 0.5 に対し，2 を累積する形で更新する（図 9(d)）．

最後に 17 行目において，再び $rd3$ を介したメモリアクセスが行われるが，このアクセスの前に 16 行目において，%$rd3$ の値が更新されている．よってこのアクセスは，先の 12 行目における %$rd3$ を介したアクセスとはアクセス対象が異なるとみなし，$wcount$ の値を別に記憶する (図 9(e))．

以上の手順により，このコードに対するアクセス先別 $wcount$ の算出を完了する．

以上のようにして算出した，各アクセス先に対する重み付きアクセス回数が多い場合は，データ再利用機会が多いと判断し，キャッシュが有効に働く LRR を採用し，そうでない場合は GTO を採用する．プログラム全体としてのデータ再利用機会の多寡は．今回の実装では $wcount$ の単純な平均値として求めることとした．この例の場合，$(1+2.5+2+1)/4 = 1.625$ となる．例えばしきい値が 1 であった場合，これを上回っているため LRR が適していると判断する．

この判断結果は，デバイスコードの冒頭部に対して今回実装として追加した専用命令によって埋め込み，実行時に利用する．

4.3 ポリシー切替部

前節で述べた PTX コードに埋め込まれた専用命令を用いて Warp Scheduler のスケジューリングポリシーをプログラムごとに変更する．3 節で述べたように，LRR と GTO の具体的な動作では，実行ユニットから Warp を追い出す指標および，Selector 内に位置する Warp Picker が実行準備が整っていない Warp をスキップした際に当該 Warp に対して Priority Rank を更新するかどうかの 2 点が異なる．よってこの 2 点を切り替えられるように既存の Warp Scheduler を拡張する．ここで，拡張後の Warp Scheduler の概略図を**図 11** に示す．プログラム中に指定されたポリシーを命令解析により検出すると，その結果を GPU 内部に保持し（図中 Policy Flag），その値に基づいて Warp Picker の動作を切り替える．まず LRR と GTO では，実行ユニットから Warp を追い出す際の基準が異なるため，LRR が選択されている場合はタイムクォンタムの時間が経過したかを判断するためにクロックカウンタの値を，GTO が選択されている場合は，Stall 検出器の出力

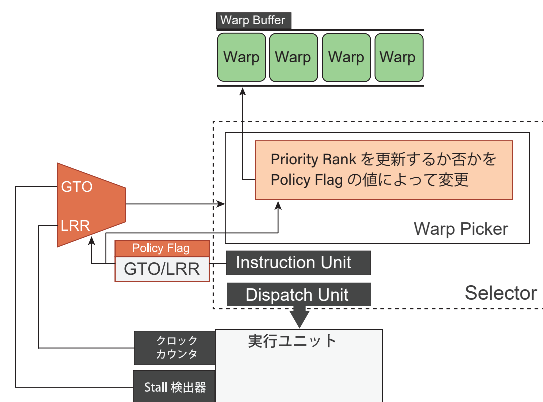

図 11 拡張後の Warp Scheduler

を，それぞれ切り替えて参照する必要がある．また 2 つのポリシーでは，Warp Picker によりスキップされた Warp に対して，Priority Rank を更新するか否かについても異なる．よって，LRR が選択されている場合は 3.1 に述べた方針に基づいた Priority Rank の更新を行うが，GTO が選択されている場合はこれを行わない，という制御を追加する．今回の実装では，プログラムごとにポリシーを切り替えるため，これらの切り替え制御が行われるのはプログラム開始時に一度のみであり，プログラムの実行時間全体に対する影響はほぼ無視することができる．なお，これらのポリシー切り替え動作はすべてプログラム実行前に行うため，スケジューリングポリシーの切り替えが実行時間に与える影響は無視できると考えられる．

5. 評価と考察

本章では，4 章で述べた提案手法を用いて評価対象プログラムを実行した結果を示す．

5.1 評価環境

4 章で述べた提案手法を用いて，プログラムの実行時間と MPKI の 2 つの項目について評価した．評価環境は 3.3 節の調査で用いたものと同様であり，使用したベンチマークプログラムは 3.3 節の調査に使用した SCP，GUPS，HISTO，SAD に加えて，Rodinia ベンチマーク [9] から HS（Hotspot）および LUD（LU decomposition），BLK（BlackScholes）[4] および SHOC ベンチマーク [10] から RED（Reduction）を使用した．

5.2 時間評価

まず，プログラムの実行時間の評価結果を**図 12** に示す．図 12 では，各ベンチマークプログラムにおける実行時間を 3 本のバーで表しており，これらのバーはそれぞれ左から GTO，LRR，提案手法の結果を表している．なお，評価結果は GTO での結果を 1 として正規化している．結果から，提案手法では多くのベンチマークで LRR/GTO の

図 12　実行時間評価

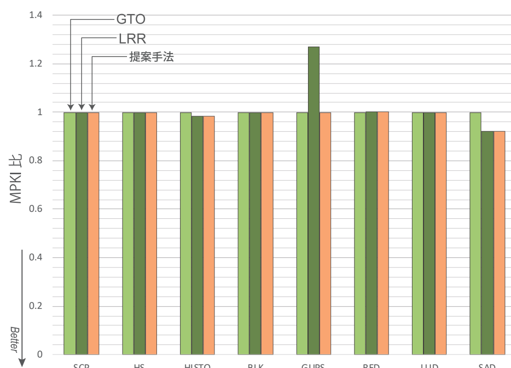

図 13　MPKI 評価

うち高速な方とほぼ同じ実行時間が達成できており，適切なスケジューリングポリシーを選択できていることがうかがえる．しかし，SCP，BLK および HS では適切なポリシーを選択できていない．

BLK はプログラム中に現れるロードストア命令の大半がループ構造内部に含まれているようなプログラムであり，今回設計した重み付けアルゴリズムに基づいて静的解析により求めた命令数と実際に実行された動的命令数とに大きな乖離が生じたことが原因であると考えられる．また，HS はロードストア命令の大半が条件分岐構造が多重に組み合わさった内部で行われるようなプログラムであり，やはり静的解析により求めたアクセス回数と実際のアクセス回数とに大きな乖離が生じたと考えられる．さらに SCP は条件分岐構造の内部にループ構造が含まれるようなプログラムであり，BLK や HS と比較してさらに複雑な構造となっているため静的解析により求めたアクセス回数と実際のアクセス回数とに大きなカイリが生じたと考えられる．そのため今後は，プログラム全体の構造を考慮に入れた閾値のチューニングや静的解析時に用いる重みを更に改良することが挙げられる．

5.3　MPKI 評価

次に，MPKI の評価結果を**図 13** に示す．図 13 では，各ベンチマークプログラムの MPKI を 3 本のグラフで表しており，これらは図 12 と同様，**LRR**，**GTO** および**提案手法**に対応している．なお，評価結果は GTO での結果を 1 として正規化している．結果より，ほぼ全てのベンチマークプログラムにおいて，提案手法は．キャッシュミス回数のより少ないポリシーと同等のミス回数を達成できていることが確認できた．注目すべき点としては RED，LUD など適切なスケジューリングポリシーを選択できていたベンチマークでもキャッシュミス回数にポリシー間で違いがないものがあり，これらのベンチマークプログラムではキャッシュミス回数以外でポリシー適否の要因があると考えられる．よって今後は，キャッシュミスミス回数の変化以外にも，速度向上に寄与する要因を更に調査し，静的解析対象の拡大や閾値の調整を行いたいと考えている．

6.　おわりに

本論文ではまず，GPU の代表的スケジューリングポリシーである LRR，GTO それぞれの特徴を述べ，キャッシュが有効に働くか否かによってプログラムごとに適したスケジューリングポリシーが異なることを実際の評価結果から示した．その上で，CUDA の PTX コードを静的解析し，ループや条件分岐によるメモリアクセスの変動を考慮して同一アドレスへの平均アクセス回数を算出し，求めた平均アクセス回数と，経験則から得られたしきい値とを比較することで，適切なポリシーをプログラムごとに選択する手法を提案した．

提案手法の有効性を検証するため，GPU の研究に広く持ちいられている GPGPU-Sim を拡張した Mafia シミュレータを用いて評価を行った．その結果，提案手法では大半のプログラムにおいて，LRR/GTO のうちより優れた性能を発揮するポリシーを適切に選択できていることを確認した．

今後の課題として，スケジューリングポリシーを判断する際に使用した閾値や今回定義した重み付け算出式の改良が挙げられる．さらにキャッシュ利用効率に加えて，別の指標を検討することが挙げられる．

参考文献

[1] NVIDIA Corp.: www.nvidia.com/.
[2] NVIDIA Corp. : *CUDA C Programming Guide*, 7.0 edition (2015).
[3] Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H. and Aamodt, T. M.: Analyzing CUDA workloads using a detailed GPU simulator, *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, pp. 163–174 (2009).
[4] Jog, A., Kayiran, O., Kesten, T., Pattnaik, A., Bolotin, E., Chatterjee, N., Keckler, S. W., Kandemir, M. T. and Das, C. R.: Anatomy of gpu memory system for multi-application execution, *Proceedings of the 2015 International Symposium on Memory Systems*, ACM, pp. 223–234 (2015).
[5] Fermi Architecture Whitepaper: www.nvidia.com/content/PDF/fermi white papers.

© 2019 Information Processing Society of Japan

[6] CUDA ToolKit 4.0: https://developer.nvidia.com/cuda-toolkit-40.
[7] Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D. and Hwu, W.-m. W.: Parboil: A revised benchmark suite for scientific and commercial throughput computing, *Center for Reliable and High-Performance Computing*, Vol. 127 (2012).
[8] PTX ISA: https://docs.nvidia.com/cuda/parallel-thread-execution.
[9] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing, *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Ieee, pp. 44–54 (2009).
[10] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V. and Vetter, J. S.: The scalable heterogeneous computing (SHOC) benchmark suite, *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ACM, pp. 63–74 (2010).

© 2019 Information Processing Society of Japan