Vol.2021-ARC-245 No.2
2021/7/20

Dualflow アーキテクチャの適用による GPU の高電力効率化の検討

松尾 玲央馬[1,2,a)]　眞下 達[3]　塩谷 亮太[1]

概要：GPU（Graphic Processing Unit）では高いデータ並列処理性能を実現できる一方で，その大きな消費電力が問題となる．一般に GPU では多くのスレッドを同時に実行することで様々なレイテンシを隠蔽し，それによって高い性能を実現している．しかし，多くのスレッドを同時実行するためにはそれらのコンテキストを保持する巨大なレジスタ・ファイルが必要であり，それが消費電力を増やしてしまっている．これに対し，本研究では Dualflow アーキテクチャと呼ぶ命令セット方式を用いることで GPU の電力効率を改善する．Dualflow アーキテクチャでは，命令はレジスタ番号ではなく命令間の距離でオペランドを指定する．このオペランドの指定方法により，Dualflow アーキテクチャでは偽の依存が原理的に生じない．この性質を利用し，本研究ではリネーム・ロジックなどの高コストな回路を使わずに GPU において out-of-order 実行を実現する．Out-of-order 実行によってスレッド内でレイテンシの隠蔽が可能となるため，レイテンシの隠蔽に必要なスレッド数を削減することができる．それによりレジスタ・ファイルのサイズを削減し，電力効率を向上させることを目指す．本稿では，この Dualflow アーキテクチャの GPU への適用方法やその課題について説明し，性能などに関して行った予備評価について述べる．

1. はじめに

近年，GPU（Graphic Processing Unit）が機械学習などを含む様々な分野でアプリケーションを高速化するために用いられている．GPU は並列処理に特化したプロセッサであり，スループットを重視する設計となっている．GPU は多数の並列に動作する計算ユニットを備える．それらにおいて多くのスレッドを起動し，マルチスレッド実行を行う事でレイテンシを隠蔽する．これにより，メモリアクセスや浮動小数点演算などの長いレイテンシを有効に隠蔽し，高いスループットを実現する．このような構造により，GPU は独立した処理を多く行うような場合に CPU (Central Processing Unit) よりも高い性能を実現することができる．

GPU は高いスループットを実現する一方で，その消費電力が非常に大きく，電力効率に問題がある．この大きな消費電力の要因の１つが巨大なレジスタ・ファイルである．前述したように GPU は多くのスレッドを同時に実行するため，それらのコンテキストを保持するための巨大なレジスタ・ファイルを持つ [1]．レジスタ・ファイルを構成する SRAM (Static Random Access Memory) は一般にその容量に比例して消費電力が増加するため，レジスタ・ファイルの消費電力は非常に大きなものとなる．この結果，レジスタ・ファイルは GPU における消費電力の大きな部分を占めている．

本研究では，スレッドあたりの性能を向上させることで GPU の電力効率を改善する．スレッドあたりの性能を向上させることでレイテンシ隠蔽のために実行されるスレッド数を削減し，結果として性能を保ったままレジスタ・ファイルを縮小する．これを実現するため，本研究では GPU に out-of-order 実行を導入する．GPU では命令を動的にスケジュールするための機構を元から備えており，これを利用することで低コストで out-of-order 実行を実現する．以下では，(1) このスケジューラについて述べた後，(2) それを利用して out-of-order 実行を実現するために導入する命令セット・アーキテクチャについて順に述べる．なお，ここでは後で詳しく述べるように NVIDIA 社の Single Instruction Multiple Thread（SIMT）アーキテクチャ [2] の GPU を前提として説明を行う．

(1) GPU は小型のリザベーション・ステーションを備えており[*1]，それにより動的な命令スケジューリングを行う事でバンク・コンフリクトなどによるレイテンシ

[1] 東京大学
[2] 日本学術振興会特別研究員 DC
[3] 名古屋工業大学
[a)] matsuo@rsg.ci.i.u-tokyo.ac.jp

[*1] Operand Collector と呼ばれる [2].

の乱れを吸収している．ただし GPU では命令の発行はあくまで in-order であり，命令間に偽の依存がある場合はそこで発行を停止する [2]．偽の依存を解消するためには一般にレジスタ・リネームが有効だが，その実現のために必要な回路は非常に高コストである．そのため GPU では通常はレジスタ・リネームは採用されていない．

(2) これに対し，本研究では Dualflow アーキテクチャ [3], [4], [5], [6] と呼ぶ命令セット方式を導入し，レジスタ・リネームを行わずに偽の依存を取り除く．Dualflow アーキテクチャは独自のオペランド参照方式を持ち，それにより偽の依存を一切生じない．このため，Dualflow アーキテクチャを用いたプロセッサではレジスタ・リネームを行う必要がそもそもない．

Dualflow アーキテクチャを導入することによって偽の依存は生じなくなるため，GPU が元から備えるスケジューラを利用する事により非常に低コストで out-of-order 実行を実現できる．なお，Dualflow アーキテクチャの導入による out-of-order 実行はレジスタ間の依存についてのみを考慮しており，メモリ・アクセス命令については別の対処が必要である．本稿では提案手法の実装方法や，モリ・アクセス命令の扱いなどの課題について説明し，これまでに行った予備評価の結果について述べる．

2. 背景

本節では，本研究の背景となる GPU の基本的なアーキテクチャと，Dualflow アーキテクチャについて順に説明する．

2.1 既存の GPU のアーキテクチャ

本節では，NVIDIA 社による GPU のアーキテクチャについて，本研究と関連が深い部分を中心に説明する．以降の説明は，より具体的には広く用いられている GPU シミュレータである GPGPU-Sim [2] が採用しているモデルに基づく．このモデルは NVIDIA 社の代表的なアーキテクチャに基づいており，シミュレーションでは実際の NVIDIA 社の GPU に近い IPC (instruction per cycle) を達成することができる [2].

2.1.1 全体像

図 1 に示すように，GPU は SIMT コア [*2]と呼ばれるプロセッサと，それらを束ねた SIMT クラスタから成る．SIMT コアはそれぞれが独立したマルチスレッド・プロセッサであり，複数スレッドが同時に動作する．各スレッドが同じ制御フローを実行している限りは SIMT コアは SIMD (Single Instruction Multiple Data) アーキテクチャと同様に，単一の命令で複数のデータを処理する．SIMD

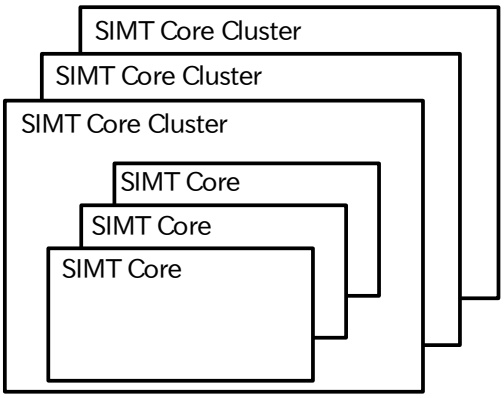

図 1 想定する GPU モデルの全体像

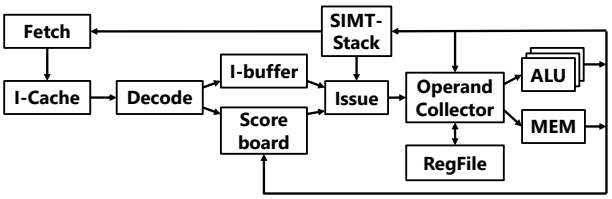

図 2 SIMT コアの構造

アーキテクチャと異なり，各スレッドが異なる制御フローを実行する際は時分割で各スレッドを処理する．これにより，SIMD アーキテクチャと同様にして演算器間で命令制御部を共有することで電力効率を高めつつ，各スレッドが独立の制御フローを実行できることでプログラマビリティを上げている．

2.1.2 SIMT コアの構成

次に，図 2 を用いて SIMT コアの構造について説明する．SIMT コアはスーパスカラ・プロセッサと似た命令スケジューリング機構を持つ．

(1) Fetch ステージでは次に実行すべきスレッドを選択し，PC を用いて命令キャッシュへとアクセスを行う．命令キャッシュから読み出した命令は Decode ステージへ送られる．

(2) Decode ステージでは命令をデコードした後，命令を **I-Buffer** へと送る．I-Buffer へと送られた命令は，オペランドがレディかどうかチェックされる．具体的には，**Scoreboard** を用いて，真のデータ依存及び偽の依存が存在しないことをチェックする．Scoreboard は，レジスタがレディかどうかを記録するためのテーブルである．

(3) オペランドが全てレディになると，命令は Issue され **Operand Collector** へ送られる[*3]．Operand Collector は，out-of-order 実行プロセッサにおけるリザベーション・ステーションと同等の機能を有する．具体的には，Operand Collector 内の命令は，オペランドがレジスタ・ファイル及び実行ステージから送られてくるのを待ち，それらが全て揃った段階で命令を実行ユ

[*2] SM: Streaming Multiprocessor とも呼ばれる．

[*3] ここでの Issue は CPU における Dispatch に対応する．

ニットへ送る．

(4) 実行ユニットで命令を実行し，実行結果をレジスタ・ファイル及び Operand Collector に送る．また, Scoreboard の更新を行う．

SIMT コアでは，単一の命令で一度に複数のスレッドの演算を行う．この単一の命令で一度に同時に演算を行う単位を **Warp** と呼ぶ[*4]．SIMT コアでは基本的に Warp 単位で命令の処理が行われる．

SIMT コアで処理中の Warp の実行状況は，**SIMT-Stack** と呼ばれるスタックで管理される．SIMT-Stack は，PC 及び Warp 内のどのスレッドを実行するかをビット・マスクで表現した**アクティブ・マスク** をエントリとして持つ．アクティブ・マスクにより，Warp 内のスレッドが異なる制御フローを持つ場合でも Warp 単位で処理を行えるようになっている．Fetch ステージでは，この SIMT-Stack の中から次に命令の読み出しを行う Warp を選択して命令のフェッチを行う．

2.1.3 Operand Collector

次に，Operand Collector について図 3 を用いて詳しく説明する．Operand Collector は，レジスタ・ファイル及び実行ユニットから送られてくるオペランドを受け取るための機構であり，out-of-order 実行機能をもつ CPU におけるリザベーション・ステーションと類似した機能を持つ．

Operand Collector が必要な理由は，次の通りである．SIMT コアでは，非常に多数の命令が同時に実行されるため，高いスループットを達成するレジスタ・ファイルが必要である．しかし，そのために単純にレジスタ・ファイルのポート数を増加させてしまうと，配線遅延が増加する上，エネルギー及び面積を悪化させてしまう．そこで，高いスループットを達成するレジスタ・ファイルをエネルギーと面積を抑えつつ構成するためにマルチバンク RAM が使用される．ただし，マルチバンク RAM では同一バンクへとアクセスが集中してしまうと，アクセス・レイテンシが増加してしまう．Operand Collector は，このレイテンシが可変なマルチバンク RAM で構成されたレジスタ・ファイルからオペランドを読み出すための機構である．

Operand Collector の構造について説明する．Operand Collector は，Collector Units と呼ばれるバッファとバンク・コンフリクトを調停するための Arbitrator で構成される．Collector Units は，各命令のオペランドの情報を保持しており，次の 3 つのエントリを持つ．(1) レジスタ番号，(2) オペランドがレディかどうかを示す 1 ビット，(3) オペランドの値．ある命令のオペランドが全てレディになった段階で，命令の情報と一緒にオペランドを実行ユニットへ送る．レジスタ・ファイルへの読み出し／書き込み要求を行う場合は，まず要求を Arbitrator へ送る．Arbitrator は，バンク・コンフリクトが起きないようにレジスタ・ファイルの各バンクへのアクセス調停を行う．

[*4] 通常，NVIDIA の GPU では Warp は 32 個のスレッドで構成される

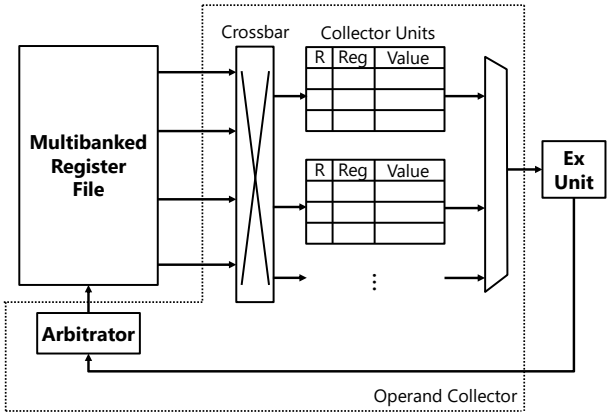

図 3 Operand Collector の構造

上記で説明した Operand Collector によるオペランド待ち合わせはスーパスカラ・プロセッサのスケジューラの動作とほぼ同じだが，Operand Collector からの命令発行は in-order に行う．これは SIMT コアではレジスタ・リネームを行っていないためであり，そのため偽の依存を無視して out-of-order に実行すると実行の正しさが保たれないためである．

2.2 Dualflow アーキテクチャ

本節では，Dualflow アーキテクチャ [4], [5] について説明する．Dualflow アーキテクチャでは，オペランドとしてデータを消費する命令（コンシューマ）ないしはデータを生産する命令（プロデューサ）を指定することにより，明示的にデータの受け渡しを行う命令セット・アーキテクチャである．Dualflow アーキテクチャのうち，プロデューサからコンシューマを指定するものを順 Dualflow，コンシューマからプロデューサを指定するものを逆 Dualflow と呼ぶ．本稿ではこれらのうち，逆 Dualflow を用いる．以下では，特に断りがない場合は Dualflow と言った場合は逆 Dualflow を指すものとする．

従来の命令セットは，図 4 (a) のようにレジスタ番号を指定することによってオペランドの指定を行う．これに対し，Dualflow アーキテクチャでは図 4 (b) のように命令間

```
I0: r1 ← load A
I1: r2 ← load B
I2: r3 ← r1 + r2
```

(a) 従来の命令セット

```
        ┌ I0: load A
        │ I1: load B
変位: 2 └ I2: add |2|, |1| 変位: 1
```

(b) Dualflow アーキテクチャ

図 4 プログラムの例

の変位（命令間距離）を用いてオペランドを指定する．この例では，命令 $I2$ は $I0$ と $I1$ の結果をソース・オペランドに用いており，$I2$ と $I0$ の命令間距離 2 と $I2$ と $I1$ の命令間距離 1 がそれぞれソース・オペランドとして指定されている．

Dualflow アーキテクチャは従来の命令セット・アーキテクチャと異なり偽の依存が生じない．Dualflow アーキテクチャではデスティネーション・レジスタは各命令につき一つが暗黙的に確保され，それが一度だけ書き込まれる．また，命令長の制約からソース・オペランドの指定に使う距離には命令セット上定義される最大値が存在し，古い命令の結果は順次参照不可能になる．このため、デスティネーション・レジスタをリングバッファから順番に割り当てていくことができ、寿命が切れたレジスタにのみ上書きが発生することが保証される．これによって Dualflow アーキテクチャでは偽の依存が存在せず，レジスタ・リネームは不要となる [3].

Dualflow アーキテクチャを実装したものには，通常の命令セットから動的に Dualflow 命令へ変換して実行する Renamed Trace Cache [5] と呼ばれるアーキテクチャと，コンパイラによって静的に生成した Dualflow 命令を実行する STRAIGHT プロセッサがある [3]．本研究では STRAIGHT プロセッサと同様に，静的に Dualflow 命令を生成して実行する．

3. Dualflow アーキテクチャの適用による GPU の電力効率の向上

本研究では，GPU へ Dualflow アーキテクチャを導入し，低コストで out-of-order 実行を実現することで電力効率を向上させる．前述したように Dualflow アーキテクチャでは偽の依存が生じない．これを利用し，GPU が元から備えるスケジューラを利用する事によって out-of-order 実行を低コストで実現できる．out-of-order 実行によってレイテンシの隠蔽のために実行されるスレッド数を削減し，性能を保ったままレジスタ・ファイルを縮小して電力効率を改善する．

以下では，3.1 節で提案手法のマイクロアーキテクチャを示し，3.2 節で out-of-order 実行によるスレッド数の削減について述べる．3.3 節で提案手法による電力削減とコストについて説明した後，3.4 節で提案手法の課題について述べる．

3.1 マイクロアーキテクチャ

GPU に Dualflow アーキテクチャを適用するためには，次に述べる 2 つのマイクロアーキテクチャの変更が必要である．変更が及ぶのは，主に図 5 で灰色で示した部分である．

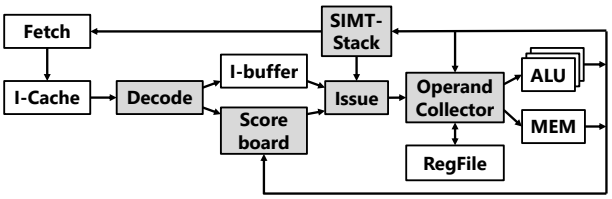

図 5 提案手法で変更が及ぶ部分

3.1.1 RP (Register Pointer) の追加

Dualflow アーキテクチャでは，命令間距離を用いてオペランドを指定するが，これを実現するためには次のようにすれば良い．レジスタ・ファイルをリング・バッファで構成し，命令が実行される毎に新たに連続領域からデスティネーション・レジスタを割り当てる．そして，ある命令のソース・レジスタに対応するインデックスは，その命令のデスティネーション・レジスタに対応するインデックスから，ソース・オペランド・フィールドで指定されている命令間距離を引くことによって計算することができる．

以上の操作を行うために，**Register Pointer** (RP) と呼ぶ特殊なレジスタをスレッド毎に追加する．RP は，そのスレッドの命令が実行されるごとにインクリメントされる．RP が指す番号は，そのまま命令のデスティネーション・レジスタのインデックスとなり，ソース・レジスタのインデックスは，RP が示す値から命令間距離を引くことによって計算される．

SIMT コアでは，スレッドの実行状態は図 5 における SIMT-Stack と呼ばれるバッファで管理されているため，このバッファに新しく RP を格納するエントリを追加する．命令を I-Buffer から Operand Collector へ Issue する時，SIMT-Stack からそのスレッドに対応する RP を読み出し，Operand Collector におけるオペランドの読み出しや，実行結果の書き込みのためのレジスタ・インデックスを計算するために用いる．RP の更新については，命令が I-Buffer から Operand Collector へ Issue される時に RP の読み出しと同時に行えば良い．Issue は in-order に行われるため，RP の読み出しが行われる度にインクリメントをすることで，命令毎に正しい RP を生成することができる．

通常のスーパスカラ・プロセッサと同様に命令の実行は out-of-order に行われるため，レジスタ・ファイルへの書き込みも投機的に out-of-order に行われる．通常のスーパスカラ・プロセッサでは例外からの回復を行うために Reorder Buffer (ROB) が必要となるが，Dualflow アーキテクチャでは例外からの回復もより簡単である．すなわち，Dualflow アーキテクチャでは RP を例外した命令の位置まで戻すだけで良い．このため，提案手法では各命令が完了したかどうかのみを追跡すればよく，これは Scoreboard とほぼ同じ回路で実現できる．

3.1.2 偽の依存のチェックの削除

Dualflow アーキテクチャでは，プログラム中に偽の依存が存在しないため，従来の GPU で行っている偽の依存のチェックが不要になる．偽の依存のチェックは図 5 における Scoreboard で行っているため，この部分で行っている偽の依存のチェックのため回路を省略する．

3.2 Out-of-order 実行によるスレッド数の削減

提案手法では out-of-order 実行により，スレッド内の命令間でレイテンシを隠蔽する能力が向上する．このため，別のスレッドによってレイテンシを隠蔽する必要性が低くなる．

これを図 6 を用いて説明する．図 6 は，命令列 $I0$〜$I2$ を従来の GPU 及び提案手法で実行した時のパイプライン図である．図内の F, D, X はそれぞれフェッチ，デコード，実行ステージを表している．命令の実行レイテンシは，ロード命令が 4 サイクル，加算命令が 1 サイクルである．命令の実行幅は 1 で，命令の実行は完全にパイプライン化されていると仮定する．この GPU は $T0$〜$T3$ の 4 つのスレッドを実行しており，そのうちの $T0$ の処理に注目する[*5]．

$I1$ は $I0$ の結果を使用するため，$I1$ は $I0$ の実行が完了するまで実行されない．そして，(a) 従来手法では，$I0$ と $I2$ の間には偽の依存が存在するため，$I0$ の実行が完了するまで $I2$ は実行されない．その間，灰色で示すように他のスレッド（$T1$〜$T3$）の命令が実行され，その後に赤で示すように $I2$ が実行される．一方，(b) 提案手法では偽の依存が存在しないため，$I2$ の実行は $I0$ の実行が完了する前に開始することができる．ただし，$I2$ が実行ステージに到達するまでは他のスレッドの命令（$T1$）を実行する必要がある．

$I0$ のレイテンシの隠蔽のために実行されるスレッドの数を比較してみると，(a) では他のスレッドが 3 つ実行されているのに対し，(b) では 1 つのスレッドしか実行されていない．このように，提案手法では out-of-order 実行によってスレッド内のレイテンシ隠蔽能力が向上するため，レイテンシの隠蔽のために必要なスレッドの数を削減できる．

3.3 電力効率の改善と追加される回路のコスト

Out-of-order 実行によるレイテンシの隠蔽を利用し，SIMT コアあたりの同時に実行するスレッド数を削減する[*6]．同時に実行するスレッド数を削減すると，スレッドに割り当てられているいくつかのハードウェア・リソースとそれに係る電力が削減できる．削減できるリソースは，図 5 におけるレジスタ・ファイル (RegFile)，I-Buffer，Scoreboard 及び SIMT-Stack である．特に，レジスタ・ファイルはエントリ数が非常に多いので（SIMT コアあたり 65536 エントリ程度），大きくリソースと電力を削減できる．

その一方で提案手法に必要な追加ハードウェア・リソースのコストは小さい．必要な追加ハードウェア・リソースは，バッファに対するフィールドの追加及び加算回路のみである．このように少ない追加リソースで out-of-order 実行が実現できる理由は，(1) レジスタ・リネーミングを行う必要がない，(2) GPU は元から命令スケジューラを備えており，これを流用している，(3) Dualflow アーキテクチャの特性により例外からの回復に ROB が必要ないためである．

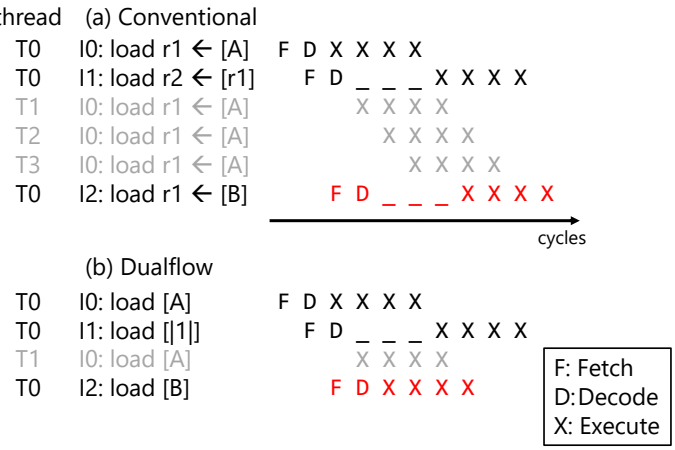

図 6 従来手法と提案手法のパイプライン図

3.4 課題

本節では，提案手法における以下の課題について順に説明する．

(1) 実行命令数の増加
(2) メモリ順序違反

3.4.1 実行命令数の増加

本節では Dualflow アーキテクチャで課題となる命令数の増加について説明する．Dualflow アーキテクチャでは距離でオペランドを指定しているため，分岐の合流時などに距離を合わせるために追加の命令が必要となる場合がある．この結果，通常の命令セットと比べて命令数が増えてしまい，実行速度が低下する場合がある．この追加命令を削減するためのコンパイラ最適化技術は発展途上であり，いくつかの手法によって，命令数を大きく削減できることがわかっている [7], [8], [9], [10], [11]．また，GPU では基本的には分岐を多く含まない小さなループを実行することが多いため，通常の CPU における実行と比べて命令数の増加による問題の影響は小さいことが期待できる．後の

[*5] なお，ここでは $T0$〜$T3$ はそれぞれ独立した Warp に所属しているものとする．
[*6] コアそのものの数を減らすのではなく，コアあたりに割り当てられるスレッド数の削減を意味する．

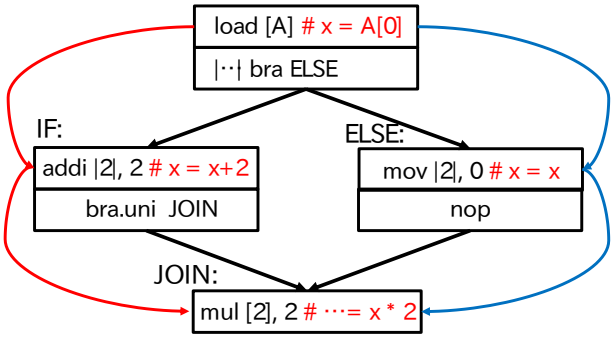

図 7 Dualflow アーキテクチャにおける分岐

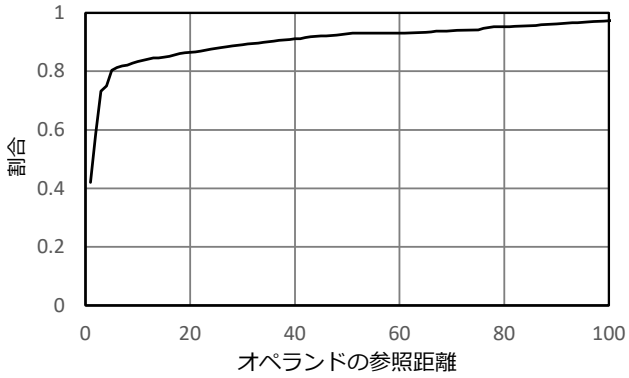

図 8 オペランドの参照距離の累積分布

4.2 節では，この命令数の増加の影響を評価する．

以下では Dualflow アーキテクチャで追加の命令を挿入する必要がある 2 つの場合について，それぞれ述べる [6] ：(1) 分岐が合流する場合，(2) 命令間距離が指定可能な最大値を上回る場合．

(1) Dualflow アーキテクチャでは，分岐命令を超えてデータの受け渡しを行う場合には注意が必要である．これを図 7 を用いて詳しく説明する．図 7 は，分岐命令によって実行パスが 2 つに分かれているが，どちらのパスを通ったとしても，コンシューマは適切なプロデューサからデータを受け取らなければならない．特に問題となるのが，分岐の合流点を超えてデータの受け渡しを行う場合である．この場合，どちらのパスを通った場合でも，コンシューマのオペランドとして，同じ変数に対応する命令を指定しなければならない．これを行うために，図 7 の ELSE のパスのように転送命令 (mov 命令) や nop 命令を挿入する必要が生じることがある．

(2) オペランドに用いる命令間距離が指定可能な最大値を上回る場合には，適宜スピルを行う必要がある．具体的には，データをいったんメモリにストアし，必要になったらメモリからロードする処理が追加で必要となる．

スピルの処理を行う回数は，指定可能な最大命令間距離を大きくすることで減らすことができる．ただし，最大命令間距離は無制限に大きくすることはできないので，現実的な値を使用する必要がある．そこで，最大命令間距離をどの程度まで大きくすれば，大部分のオペランドをカバーすることができるかを調べるために予備評価を行った．図 8 は，4.1 節で述べる評価方法で Rodinia ベンチマーク [12] の nw を実行した際のオペランドの参照距離の累積分布である．同図によると，オペランドの参照距離が 5 より小さい場合が全体の 8 割を占めており，参照距離 40 以下の場合になると全体の 9 割を占める．他のベンチマークでも同様に参照距離が小さい場合が支配的であるという結果となった．したがって，最大命令間距離をそこまで大きな値に設定しなくても，スピルを行わなければならない頻度は小さいと言える．

3.4.2 メモリ順序違反

提案手法では，メモリ命令も out-of-order に実行するため，メモリ順序違反を起こす可能性がある[*7]．このため実行の正しさを保証するためには，メモリ順序違反が生じた際にそれを検出して回復するか，あるいは何らかの方法によりメモリ・アクセスの正しい実行順序を保証する必要がある．通常のスーパスカラ・プロセッサでは，メモリ順序違反の検出と回復のためにロード・ストアキューやロードの再実行機構を取り入れているが，しかしハードウェア・コストや電力の制約が厳しい GPU ではこれらの手法を導入することは現実的ではない．

以下では，低コストで正しいメモリ・アクセスの順序を保証する 2 つの方法を提案する．なお，4 節の評価ではこれらの機構は未実装であり，メモリ順序違反が起きないと理想化した上で評価を行っている．

(1) バリア命令を使った方法：この方法では，メモリ順序違反を起こす可能性があるメモリ命令の間にバリア命令を挿入することで，これらの命令の実行順序が入れ替わらないように保証する．この方法は非常にシンプルであり，既存のバリア命令を使用すれば追加コストなしで実現できる．ただし，バリア命令によって他の命令の実行順序の入れ替えが制限されてしまうため，これが性能を低下させる可能性がある．

(2) メモリ命令のカラーリングによる方法：この方法では，メモリ順序違反を起こしうるメモリ命令に静的にカラーリングを施し，同じグループに属するメモリ命令同士は in-order に実行する．これは具体的には，グループ番号をメモリ・アクセス命令に埋め込み，おなじグループ番号を持つ命令間にレジスタ間と同様の依存があるものとしてスケジューリングすることにより実現する．この手法はバリア命令を使った方法よりも複雑にはなるが，メモリ順序違反を起こしうるメモリ命令と無関係な命令の実行順序の入れ替えを制限しな

[*7] 既存の GPU ではメモリ命令は in-order に実行されるため，メモリ順序違反を起こすことはない．

表 1　GPU の基本構成

Streaming Multiprocessors	68 SMs, 1440 MHz
Warp scheduler/Core	4 (GTO)
Threads/Core	2048
Registers/Core	65536
Collector Units/Core	8 entries
Shared Memory/Core	128 KB
L1 Data Cache/Core	64KB, 8-way, LRU, 128 B/line
L2 Data Cache	4MB in total (64 slices, 16-way)
Memory	100 cycle-latency 1,523 MHz
Interconnection Network	1440 MHz

いため，バリア命令を使った方法よりも性能が高くなることが期待できる．こちらも方法では，コンパイラ最適化やアノテーションにより適切なカラーリングを行う必要があり，プログラミングのためのコストは増加する．

4. 評価

本研究では，予備評価として，提案手法を適用し，GPU で out-of-order 実行を行うことによって得られる性能向上を測定した．4.1 節では，測定に用いた評価環境についての述べ，4.2 節で性能向上率及び実行命令数の増加率について述べる．

4.1 評価環境

本評価では，広く用いられている GPU シミュレータである GPGPU-Sim [2] を修正して評価を行った．性能の評価に用いた GPU の基本構成は RTX 3080 をベースにしており，その構成を表 1 に示す．評価モデルは，次に示す 2 種類を用いた．

- BASE: ベースラインとなる GPU.
- D-FLOW: GPU に Dualflow アーキテクチャを適用したモデル.

ベンチマーク・プログラムは，Rodinia ベンチマーク [12] の一部を使用した．BASE と D-FLOW の違いは命令セットの違いとそれによる out-of-order 実行の有無のみで，同時に実行するスレッドの数やコア数などのその他のパラメータはすべて同一で表 1 の通りである．また，評価したベンチマークはすべて規模の小さいマイクロベンチマークであるため，どちらの GPU も各プログラム中のすべてのスレッドを同時に起動することができる．これにより，DualFlow アーキテクチャと out-of-order 実行によって得られる性能向上を測定した．D-FLOW のスレッド数を減らした際の性能への影響の評価は，今後の課題である．

ベンチマーク・プログラムの Dualflow 形式への変換は，自作のスクリプト及びハンド・アセンブリを組み合わせて行った．シミュレーションは，プログラムの全区間について行った．ただし，3.4.2 節で述べたように，今回の測定ではメモリ依存予測について理想化を行っており，メモリ順序違反が起きないようになっている．

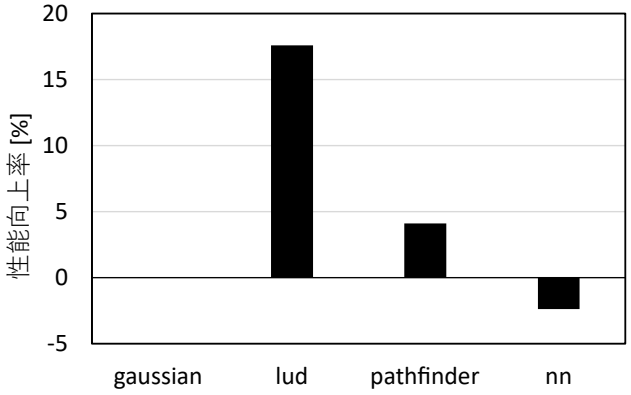

図 9　性能向上率

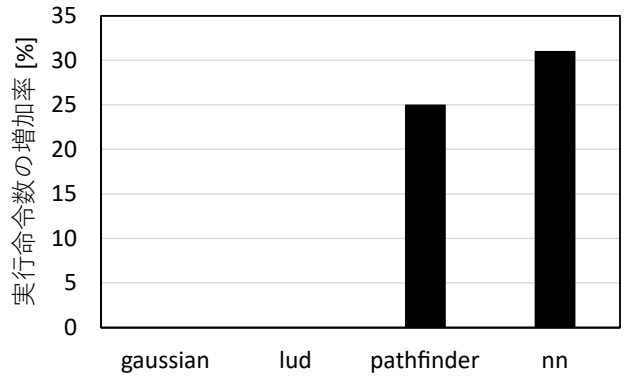

図 10　実行命令数の増加率

4.2 性能及び実行命令数の増加

図 9，図 10 に，BASE に対する D-FLOW の性能向上率及び実行命令数の増加率のグラフをそれぞれ示す．今回測定した 4 つのベンチマークで，それぞれ異なる傾向が見られた．以下では，各ベンチマークの結果について述べる．gaussian は，性能及び実行命令数共に変化が見られなかった．これは，元の GPU プログラム中に偽の依存も分岐もないためと考えられる．lud は，実行命令数は変化しなかったものの，性能が大きく向上している．これは，lud には分岐による合流が存在しないものの，偽の依存が存在していたためと考えられる．pathfinder は，実行命令数が 25% と大きく増加しているものの，性能も向上している．これは，out-of-order 実行による性能向上が，Dualflow アーキテクチャの適用による実行命令数の増加の影響を上回ったケースである．nn は pathfinder よりも実行命令数が大きく増加し，性能がベースラインと比べて低下してしまっている．これは，実行命令数の増加の影響が out-of-order 実行による性能向上を上回ってしまったケースである．

今回の評価では，4 つ中 2 つのベンチマークで性能が向上しているが，一方で nn のように性能が低下してしまうケースもある．このようなケースを減らすために，Dualflow 形

式のプログラムの最適化 [7], [8], [9], [10], [11] を行って実行命令数を減らし，性能をより向上させることが重要だと言える．

5. 今後の課題

5.1 スレッド数を削減した場合の評価

今後，本稿で実施しなかった，スレッド数を減らした際の提案手法の性能評価を行う．そのためには，今回用いたような規模の小さいマイクロベンチではなく，GPU がすべてのスレッドを同時に起動できないほど規模の大きいベンチマークを使用する必要がある．

5.2 電力評価

3.3 節で，提案手法は最大駆動スレッド数を減らすことによって電力が削減できることを述べたが，どの程度電力を削減できるかを定量的に評価する必要がある．これは実際に電力シミュレーションを行わなければ分からない．その測定を行うことが今後の課題である．

5.3 シミュレータで理想化されている部分の評価

4.1 節で述べたように，今回の測定ではメモリ依存予測について理想化を行った上で評価を行っている．しかし，提案手法の公平な評価を行うためには，実際の実装に即して評価を行うべきである．そのため，この理想化されている点については，理想化を行わずに実装を修正して評価を行うことが今後の課題となる．

5.4 プログラムの最適化

本稿における評価は，自作スクリプトによって GPU プログラムを Dualflow 形式に変換して行ったが，生成される Dualflow 形式のプログラムの最適化が不十分であり，実行命令数が増えすぎてしまうという問題がある．Dualflow 形式のプログラムの最適化は，STRAIGHT アーキテクチャの研究において盛んに行われており，冗長な転送命令の削除やループの最適化，積極的スピルなどを行うことによって実行命令数を大きく削減できることが報告されている [7], [8], [9], [10], [11]．しかし，本研究が使用した Dualflow 形式への変換スクリプトは単純な変換のみ行い，最適化はほとんど行っていない．そのため，上述した最適化手法を取り入れることで実行命令数を大きく減らすことができると考えられる．また，専用命令の追加によって実行命令数を削減する方法も考えられる．そこで，Dualflow 形式のプログラムの最適化を行った上で，再評価を行うことが今後の課題である．

6. 関連研究

本研究と同様に Dualflow アーキテクチャを導入した手法に STRAIGHT アーキテクチャがある [3]．STRAIGHT アーキテクチャは，out-of-order スーパースカラ・プロセッサに Dualflow アーキテクチャを適用することによって，パイプラインからレジスタ・リネーム・ステージを取り除く．これにより，次の 2 つの利点が得られる：(1) 通常，out-of-order 実行を行うためには，レジスタ・リネーミングを行う必要があり，これを行うためにはポート数が大きな RAM が必要になる．しかし，Dualflow アーキテクチャの導入によってレジスタ・リネーム・ステージが取り除かれるので，この RAM が不要になる．一般に多ポートの RAM は消費電力が非常に大きいため，この RAM が取り除かれることによって消費電力を改善することができる．(2) レジスタ・リネーム・ステージが取り除かれることで，フロントエンドのパイプラインが短くなり，分岐予測ミス・ペナルティが小さくなる．これにより，従来のアーキテクチャと比べて性能を向上させることができる．

STRAIGHT アーキテクチャは，Dualflow アーキテクチャを CPU に適用した手法であるが，本研究の着眼点は Dualflow アーキテクチャを GPU に適用する点にある．STRAIGHT アーキテクチャは Dualflow アーキテクチャを，レジスタ・リネーム・ステージを取り除くために活用した．それに対し，本研究では元々レジスタ・リネーミングを行っていない GPU に Dualflow アーキテクチャを適用することで，GPU で完全な out-of-order 実行を行えるようにする．この点が本研究の提案手法と STRAIGHT アーキテクチャの異なる点であり，本研究の独自な部分となっている．

7. まとめ

本稿では，GPU に Dualflow アーキテクチャを適用することで GPU の電力効率を改善する手法を提案した．一般に GPU では多くのスレッドを同時に実行することで様々なレイテンシを隠蔽し，それによって高い性能を実現している．しかし，多くのスレッドを同時実行するためにはそれらのコンテキストを保持する巨大なレジスタ・ファイルが必要であり，それが消費電力を悪化させている．本研究では，スレッドあたりの性能を向上させることで GPU の電力効率を改善する手法を提案した．スレッドあたりの性能を向上させることでレイテンシ隠蔽のために実行されるスレッド数を削減し，結果として性能を保ったままレジスタ・ファイルを縮小する．また，これを実現するために，GPU に DualFlow アーキテクチャを適用し，out-of-order 実行を行う手法を提案した．

予備評価では，GPU で out-of-order 実行を行うことによって得られる性能向上を測定した．その結果，4 つ中 2 つのベンチマークでは性能向上が見られ， out-of-order 実行によって性能を向上させられることを確認できた．一方で，1 つのベンチマークでは性能低下が見られたため，よりプログラムの最適化を行って実行命令数を削減し，より

性能を向上させることが課題となる．

謝辞 本研究は JSPS 科研費 JP20J23642，JP20H04153 の助成を受けたものである．

参考文献

[1] Jeon, H., Ravi, G. S., Kim, N. S. and Annavaram, M.: GPU Register File Virtualization, *Proceedings of the 48th International Symposium on Microarchitecture*, p. 420–432 (2015).

[2] Lew, J., Shah, D., Pati, S., Cattell, S., Zhang, M., Sandhupatla, A., Ng, C., Goli, N., Sinclair, M., Rogers, T. and Aamodt, T.: Analyzing Machine Learning Workloads Using a Detailed GPU Simulator (2018).

[3] Irie, H., Koizumi, T., Fukuda, A., Akaki, S., Nakae, S., Bessho, Y., Shioya, R., Notsu, T., Yoda, K., Ishihara, T. and Sakai, S.: STRAIGHT: Hazardless Processor Architecture Without Register Renaming, *51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 121–133 (2018).

[4] 五島正裕，グェンハイハー，縣　亮慶，森眞一郎，富田眞治：Dualflow アーキテクチャの提案，*JSPP2000*, pp. 197–204 (2000).

[5] 一林宏憲，塩谷亮太，入江英嗣，五島正裕，坂井修一：逆 Dualflow アーキテクチャ，情報処理学会論文誌コンピューティングシステム (ACS)，Vol. 1, No. 2, pp. 22–33 (2008).

[6] 入江英嗣, 山中崇弘, 佐保田誠, 吉見真聡, 吉永　努：もし ILP プロセッサのレジスタファイルが分散キーバリューストアになったら，研究報告システム・アーキテクチャ（ARC），Vol. 2013-ARC-206, No. 5, pp. 1–10 (2013).

[7] 中江哲史，入江英嗣，坂井修一：STRAIGHT コンパイラにおける不要コードの削減手法の検討，研究報告システム・アーキテクチャ（ARC），Vol. 2016-ARC-221, No. 5, pp. 1–6 (2016).

[8] 福田晃史，中江哲史，入江英嗣，坂井修一：STRAIGHT アーキテクチャにおけるループ内ロード命令の削減手法，研究報告システム・アーキテクチャ（ARC），Vol. 2017-ARC-225, No. 3, pp. 1–6, (2017).

[9] 小泉　透，中江哲史，福田晃史，入江英嗣，坂井修一：STRAIGHT 向けコンパイラによる冗長転送命令の削減，研究報告システム・アーキテクチャ（ARC），Vol. 2018-ARC-231, No. 2, pp. 1–10 (2018).

[10] 中江哲史，小泉　透，杉田　脩，入江英嗣修一：STRAIGHT コンパイラにおけるループおよび関数呼び出し最適化の評価，研究報告システム・アーキテクチャ（ARC），Vol. 2020-ARC-241, No. 5, pp. 1–6 (2020).

[11] 小泉　透，杉田　脩，塩谷亮太，入江英嗣，坂井修一：実用プログラムの STRAIGHT アーキテクチャにおける特性の解析，研究報告システム・アーキテクチャ（ARC），Vol. 2020-ARC-242, No. 1, pp. 1–6 (2020).

[12] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H. and Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing, *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54 (2009).