

UVM利用時のPTX命令順序最適化手法と 性能向上の限界に関する調査

林 大瑛¹ 巽 友佑¹ 小泉 透¹ 津邑 公暁¹

概要：GPUは簡素な設計の実行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、高速に並列計算を行なっている。近年では、多くの研究分野でGPUをアクセラレーションコアとして利用する機会が増加している。従来のCUDAの実行モデルでは、ホスト側とデバイス側の両方でメモリを確保し、カーネルを実行する前後でデータをホスト・デバイス間で明示的に転送する必要があった。近年ではホストメモリとデバイスメモリに統一的にアクセス可能とする仮想アドレス空間UVMを利用することができる。この機能を利用することでプログラミングの抽象度を高めることが可能となった。しかし、UVMを使用するとトレードオフとしてページフォルト処理が発生する。ページフォルト処理が発生した場合、20 μ s以上を要しオーバヘッドとなる。そこで本論文では、一般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をターゲットとし、ページフォルト処理を集約できるようPTX命令列を最適化する手法を提案する。提案手法では、データ依存関係を考慮しロード命令を可能な限り前倒しして連続的に配置する。また、ロード命令にデータ依存する命令は後ろ倒しすることでロード命令とデータ依存関係がある命令でストールすることを可能な限り遅くする。提案手法の有効性を検証するため、TITAN RTXを搭載した実行環境でRodinia-3.1を使用して評価を行った。提案手法を適用した結果、NVCCコンパイラが生成したPTXと比較して、ページフォルト処理回数では最大14.5%、ページフォルト発生回数では最大19.3%の削減を達成した。

1. はじめに

GPU (Graphics Processing Unit) は元来、画像処理に特化したプロセッサとして開発された。一般に、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。この特徴を活かし、GPUは簡素な設計の実行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。一方で、GPUの実行ユニットは、ハイエンドCPUに標準搭載されている投機実行やレジスタ・リネーミング等の機能を備えていない。その代わり、GPUでは多数の演算器とレジスタを持っている。また、GPUは単一の命令で多数のデータを一斉に処理することができる。この特性によりデータ並列性が高い処理ではGPUはCPUより高い演算スループットと電力効率を実現している [1]。このようなアーキテクチャ特性が注目され、GPUを画像処理以外の目的に応用するGPGPU (General Purpose computing on GPU) が発展してきている。近年では、GPUを用いてビッグデータ解析 [2] や機械学習 [3] など

を高速化する研究も多数行われている。特にHPC (High Performance Computing) 分野では、TOP500 [4] においてGPUを複数台搭載した計算機同士を高速なネットワークで繋いだクラスタ構成のスーパーコンピュータ [5] が上位にランクインするなど、GPUは現代の科学技術計算において不可欠な存在となっている。

近年CPUとGPUが統一的にメモリ空間を共有する機能としてUnified Virtual Memory (UVM) やHeterogeneous Memory Management (HMM) などが存在する [6, 7]。UVMはNVIDIA独自の機能であり、NVIDIA製GPU専用の機能である。HMMはLinuxカーネルの標準機能を活用した機能であり、AMD社のGPUなどで使用可能である。各社名称は異なるがここでは最も有名なUVMについて説明する。UVMを利用することでホスト・デバイス間でデータ通信命令の記述が省略可能となり、GPUプログラミングにおけるメモリ管理の抽象度を高めている。また、GPUで扱うデータサイズは年々増え続け、近年ではGPUのVRAM容量を凌駕するデータサイズを扱うことも多い。このような場合、UVMを利用しないとデータ依存関係を考慮しデータを分割しなければならない。その一方、UVMを利用する場合データ分割の

¹ 名古屋工業大学
Nagoya Institute of Technology

必要がなく、プログラミングが容易になる。

このように、プログラミングの抽象度を高める利益がある一方、問題点も存在する。それが UVM を用いた GPU プログラムでデータにアクセスする際に発生する可能性があるページフォルトである。ページフォルトとはデバイスのメモリアクセス時に物理メモリ上にページ割り当てがないと発生する例外である。デバイス側のメモリアクセスでページフォルトが発生した場合、20 μ s 以上を要するページフォルト処理が行われる。ページフォルト処理は大きなレイテンシを伴う処理であるため、コアがストールする可能性が高く GPU のスループットを著しく低下させるオーバヘッドとなる。このオーバヘッドを抑制するために、GPU では複数のページフォルト処理を集約して実行するよう設計されている。この問題に対しては、GPU 上に追加実装したハードウェア回路を利用しこのオーバヘッド中に、SM に割り当てられているスレッドブロックを切り替えページフォルトを追加発生させる手法が提案されている [8]。

しかし、この先行研究はハードウェアからのアプローチであり、追加ハードウェアが必要になるため市販の GPU では効果が得られない。そのため、市販の GPU でもページフォルト処理のオーバヘッドを抑制するための研究としてソフトウェアからのアプローチを模索する。そこで本論文では、一般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) [9] をターゲットとし、現状の NVCC コンパイラでは制御できていないデータ依存関係に考慮した PTX の命令順序そして命令順序をチューニングする設計を提案する。

以下、2 章では GPU のプロセッサ構造、CUDA 実行モデルについて説明する。3 章では CUDA プログラミングの進化、ページフォルト処理の概要を示す。そして 4 章で提案する最適 PTX 命令順序の方針と設計について説明する。5 章で提案手法の有効性を確認するための評価を行う。最後に、6 章で結論を述べる。

2. NVIDIA 製 GPU と CUDA

本章では、NVIDIA 製 GPU の基本構造および CUDA の実行モデルについて説明する。

2.1 GPU のプロセッサ構造

GPU は多数の演算器を並列に動作させることで高い演算性能を実現する。NVIDIA 製の GPU のプロセッサ構成の概略図を図 1 に示す。

NVIDIA の GPU は階層的なプロセッサ構成を採用している。GPU 内部には多数の SM (Streaming Multiprocessor) と呼ばれる計算ユニットが存在する。そして、SM の内部に個々の演算を行う多数の実行ユニットが存在する。

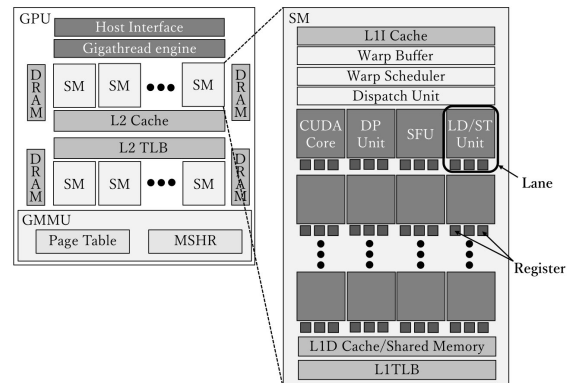


図 1: NVIDIA 製 GPU の構成図

SM は多数の実行ユニットに加え、キャッシュ、L1TLB、命令群が待機する Warp Buffer、Warp Buffer から命令群を選択する Warp Scheduler、および Warp Scheduler が選択した命令群を各実行ユニットへ割り当てる Dispatch Unit から構成されている。そして、SM 内部に存在するこれらの実行ユニットには、レジスタ (Register) と呼ばれる高速な記憶回路がそれぞれユニットごとに独立して併設されている。この実行ユニットとレジスタとを合わせてレーン (Lane) と呼ぶ。また、実行ユニットには、演算ユニットである 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 がある。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 は単精度浮動小数点数 (FP32) の積和演算や、32 bit 整数 (INT32) の加減算と論理演算を実行するユニットである。

図 1 に示すように、GPU は、各階層にアドレス変換を高速化する TLB を搭載している。各 SM は L1TLB を使用し、GPU 全体は L2TLB を使用する。L1TLB は各 SM に割り当てられた全てのスレッドで共有され、それらのスレッドがアドレス変換を要求する命令を実行した際に最初に参照される。L2TLB は SM 外に存在し、GPU 全体の全てのスレッドで共有される TLB である。L1TLB でミスが発生した場合に L2TLB が参照され、ヒットした場合、L1TLB にエントリを追加し、その物理アドレスを返す。L1TLB と L2TLB の両方でミスが発生した場合、仮想アドレスを物理アドレスに変換する GMMU (GPU Memory Management Unit) が VRAM 上を探しページテーブルエントリを取得する。そして、L1TLB と L2TLB の両方にエントリを追加する。なお、VRAM 上のページテーブルにエントリが存在しない、またはエントリが invalid である場合、データがデバイスメモリに存在しないことに起因するページフォルトとして、Miss Status Handling

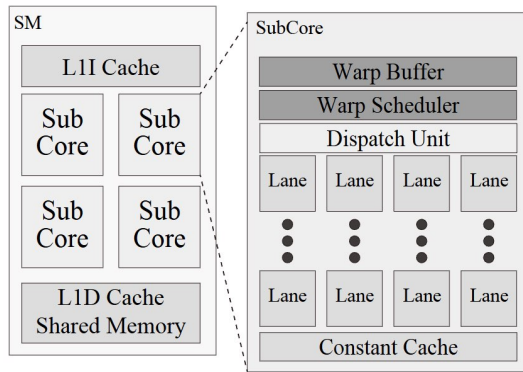


図 2: SubCore を搭載した SM の構成図

Registers (MSHR) に登録される。MSHR に登録されたページは PCIe 経由で転送されるようにスケジュールされ、データがホストからデバイスメモリに転送される。その後、Load/Store Unit に通知され、再度メモリアクセスに関する一連の処理が実行される。この方式のことを replay 方式と呼ぶ。

Volta アーキテクチャ [10] 以降の GPU では、一度により多くの命令群を実行ユニットへ割り当て可能とするために、**SubCore** [11] と呼ばれる構造を採用している。ここで、SubCore を搭載した GPU の SM の構造を図 2 に示す。Volta アーキテクチャ以降の GPU では、この SubCore を SM 内に 4 基搭載しており、SubCore は独立にスケジューリングされる。各 SubCore 内の Warp Scheduler は、同 SubCore 内の Warp Buffer から Warp を 1 つ選択して実行ユニットへ割り当てるため、SM あたり同時に最大 4 個の Warp を実行ユニットへ割り当て可能である。SubCore 内部には、Warp Buffer、Warp Scheduler、コンスタントキャッシュ、および多数の実行ユニットがある。一方で、SubCore 外部には、一次命令キャッシュ (L1I)、一次データキャッシュ (L1D) および共有メモリ (Shared Memory) があり、これらのメモリは SubCore 間で共有される。

2.2 CUDA 実行モデル

NVIDIA 製の GPU では CUDA というプラットフォームがサポートされている。CUDA は NVIDIA 製 GPU 専用のプラットフォームであり、計算に特化したプログラミングモデルである [12]。CUDA の実行モデルでは、GPU 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**、Thread の集合を **Thread Block**、Thread Block の集合を **Grid** と呼び、Grid は GPU に、Thread Block は SM に、Thread は Lane に割り当てられる。

CUDA プログラミングモデルにおいて、グリッドおよびブロックは 3 次元で管理されている。そして、グリッドは GPU 全体に、ブロックは GPU 内の SM に、スレッドは SM 内の Lane に、それぞれ対応している。なお、それ

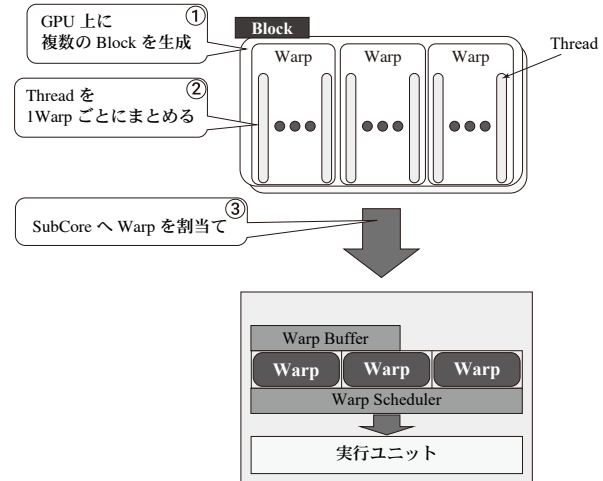


図 3: CUDA 実行モデル

ぞれのグリッド、ブロック、スレッドは固有の ID を持ち、CUDA プログラム内でこの ID を用いることで、処理単位を GPU プロセッサの各階層に割り当てることが可能である。

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。プログラマはデバイスコードを、各スレッドが行う処理を記述した関数として定義する。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多数のスレッドに割り当て、同じ命令を別々のデータに対して実行する。この実行方式のことを **SIMT (Single Instruction Multiple Threads)** と呼んでいる。SIMT において、同時に同じ命令を実行するスレッド群の 1 単位を **Warp** と呼び、1 つの Warp は 32 本のスレッドからなる。この Warp は、Warp Scheduler によって SM 上のレーンに割り当てられ、カーネル関数を実行する。図 3 は GPU 上で CUDA プログラムが実行される様子であり、図 4 は GPU 上でスレッドがレーンに割り当てられる様子である。これらを用いて、CPU が CUDA プログラムの実行を開始してから、SubCore を搭載した GPU 上で Warp 内の各スレッドがカーネル関数を実行するまでの流れを説明する。

CPU がカーネル関数を呼び出すと、GPU 上にスレッドを内包したスレッドブロックが複数生成される (図 3 中①)。そして、SM にスレッドブロックが割り当てられると、スレッドブロック内のスレッドは 32 スレッドごとに Warp にまとめられる (図 3 中②)。各 SubCore 内に割り当てられる Warp 数ができる限り等しくなるように、Warp は各 SubCore 内の Warp Buffer へ送られる (図 3 中③)。

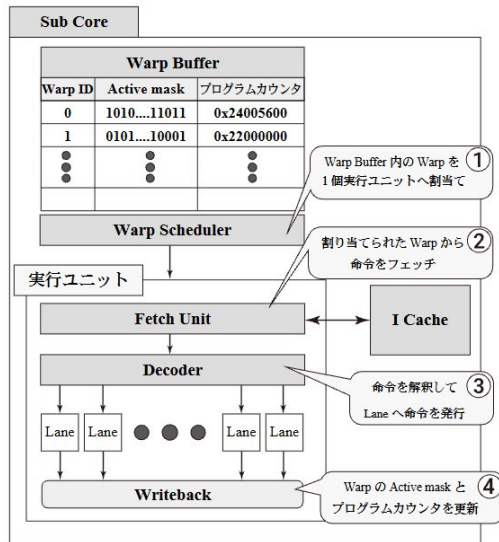


図 4: GPU パイプライン

GPUでは実行ユニットが命令を実行する際に生じるレイテンシを即時に他の命令を実行することで隠蔽し、並列処理のスループットを向上させている。SubCore内のWarp Schedulerは、1サイクルごとに、Warp Buffer内から実行可能なWarpを1つ選択する(図4中①)[13]。ロード命令やストア命令などの可変遅延命令はメモリサブシステム側で非同期に実行される。そのため、Warp Schedulerは可変遅延命令の完了を待たずに次の命令を割り当てる。また、各Warpには、Warp IDと呼ばれる識別用のID、アクティブマスク(Active mask)と呼ばれるビット列、プログラムカウンタが関連付けられている。アクティブマスクの各ビットは、各スレッドに対応しており、セットされているビットに対応するスレッドが処理される。このアクティブマスクは、Warpが初めて生成された際に、全てのビットがセットされた状態で初期化される。Warp SchedulerがWarpを実行ユニットへ割り当てたのち、Warpに関連付けられているプログラムカウンタで命令キャッシュへアクセスし、命令をフェッチする(図4中②)。その後、命令を解釈しアクティブマスクにより指定されるスレッドに限定して、レーンへ命令を発行する(図4中③)。ライトバック段階では、Warpに含まれるプログラムカウンタとアクティブマスクが更新され、Warp内に後続命令がある場合、そのWarpは再び実行ユニットへ割り当て可能となる。もし、Warp内の全ての命令が実行完了した場合やWarpがストールした場合は他の実行可能なWarpが割り当てられる(図4中④)。このように、Warp SchedulerがWarpを実行ユニットへ割り当てることで、GPU内でカーネル関数が実行される。

Volta世代以降のGPUでは、Warpを構成するスレッドを独立に実行する機能をサポートしている[10]。これは、分岐を要因とするデッドロックを回避するためであ

る。Pascal世代以前のGPUは、Warpを構成するスレッドの進行度を積極的に収束させて並列性を高める実行方法を採用していた。しかし、Warpを構成するスレッドが分岐する場合、分岐内の処理が完了するまで、もう一方の分岐内の処理を割り当てることができず、分岐内の処理が原因でデッドロックが発生する可能性があった。また、デッドロックを回避するためには並列アルゴリズムに関する知識やGPUの構造、CUDAの実行モデルを理解する必要があった。そこで、Warpを構成するスレッドを独立に実行することで、これらの問題を解決している。

ソースコード 1: CUDA C++カーネルコード

```
1 extern "C" __global__ void
2 kernel(float *a,
3         const float *b,
4         const int *c,
5         const float *d,
6         int n) {
7     int i = blockIdx.x
8             * blockDim.x
9             + threadIdx.x; // スレッドIDを計算
10
11     if (i < n) {
12         a[i] = b[c[i]] + d[i];
13     }
14 }
```

2.3 ページフォルト後にSMがストールするタイミング

ここで、GPUでページフォルトが発生した際の動作を確認する。CUDA C++のカーネルコードであるソースコード1を用いて実際の実行例を示す。ソースコード1をNVCCコンパイラによりPTXに変換し、clock64レジスタを用いてGPUの現在のサイクル数を取得できるようそのPTXを改変したコードをソースコード2に示す。

ソースコード2を実行した結果を図5に示す。図5の1~3行目はソースコード2の12, 17, 24行目にそれぞれ対応している。図5の4, 5行目はロード命令、依存関係のある命令の実行サイクル数を示している。実行結果より(1)と(2)で得られたクロック値の差は数サイクルと小さかったが、(2)と(3)で得られたクロック値の差は約70万サイクルと大きかった。このことから、ロード命令でページフォルトが発生したとしてもそれとデータ依存関係のない後続命令は実行されることが確認できた。一方、そのロード命令とデータ依存関係のある命令はストールしてしまうことも確認できた。この実行結果はGPUがTLBミスを遅延メモリオペレーション、ページフォルトを超遅延メモリオペレーションと扱っていることと整合する。

ソースコード 2: GPU サイクル測定用 PTX

```

1 //省略
2 cvta.to.global.u64      %rd5, %rd3;
3 mul.wide.s32            %rd6, %r1, 4;
4 add.s64                 %rd7, %rd5, %rd6;
5 ld.global.u32           %r6, [%rd7]; //c[i]
6 cvta.to.global.u64      %rd8, %rd2;
7 mul.wide.s32            %rd9, %r6, 4;
8 add.s64                 %rd10, %rd8, %rd9;
9 cvta.to.global.u64      %rd11, %rd4;
10 add.s64                 %rd12, %rd11, %rd6;
11
12 mov.u64 %rd20, %clock64; // (1) LD 命令実行直前
13
14 ld.global.f32           %f1, [%rd12]; //d[i]
15 ld.global.f32           %f2, [%rd10]; //b[c[i]]
16
17 mov.u64 %rd20, %clock64; // (2) LD 命令実行直後
18 //依存関係のある
19 //命令実行直前
20
21 add.f32                 %f3, %f2, %f1; //依存関係のある
22 //命令
23
24 mov.u64 %rd21, %clock64; // (3) 依存関係のある
25 //命令実行直後
26
27 cvta.to.global.u64      %rd13, %rd1;
28 add.s64                 %rd14, %rd13, %rd6;
29
30 st.global.f32           [%rd14], %f3; // a[i]
31 //省略

```

1	(1)		259176803
2	(2)		259176820
3	(3)		259903980
4	LD	: (2) - (1)	17 cycles
5	Dependencied	: (3) - (2)	727160 cycles

図 5: GPU サイクル測定用 PTX 実行結果

3. UVM とページフォルト処理の影響

本章では、従来の CUDA プログラミングと UVM を利用した際の CUDA プログラミングを比較する。そして、UVM を利用した際に発生するページフォルト処理について説明する。

3.1 CUDA プログラミングの進化

従来の CUDA の実行モデルでは、ホスト側とデバイス側の両方でメモリを確保する必要があった。CUDA C++ コードで記述された GPU プログラムのホストコードであるソースコード 3 を例に挙げると、ソースコードの 2 行目と 5 行目からその様子を確認できる。また、カーネルを実行する前に、プログラマはデータをホスト側からデバイス

側に明示的に転送する必要があった（ソースコード 3: 7 行目）。その上で、デバイス側で処理したデータをホスト側で扱う場合、デバイス側からホスト側にそのデータを返送する必要があった（ソースコード 3: 15 行目）。この実行モデルには 3 つの課題がある。1 つ目は、一般にホスト・デバイス間のデータ転送とカーネル実行は直列化しているということである。2 つ目は、GPU の実メモリより大きなメモリ領域を必要とする場合、オーバサブスクリプションを回避する方法をプログラマが構築しなければならないことである。3 つ目は、GPU メモリを確保するコードやデータを転送するコードをプログラマが逐一記述する必要があるのである。

ソースコード 3: CUDA C++ のホストコード

```

1 int *host_x, *device_x;
2 host_x = (int *)malloc(sizeof(int) * SIZE);
3 set_data(host_x, SIZE);
4
5 cudaMalloc((void **)&device_x,
6            sizeof(int) * SIZE);
7 cudaMemcpy(device_x,
8            host_x,
9            sizeof(int) * SIZE,
10            cudaMemcpyHostToDevice);
11
12 kernel<<<GRID,BLOCK>>>(SIZE, device_x);
13
14 cudaDeviceSynchronize();
15 cudaMemcpy(host_x,
16            device_x,
17            sizeof(int) * SIZE,
18            cudaMemcpyDeviceToHost);

```

ソースコード 4: CUDA C++ の UVM を利用したホストコード

```

1 int *uvm_x;
2
3 cudaMallocManaged((void **)&uvm_x,
4                    sizeof(int) * SIZE);
5 set_data(uvm_x, SIZE);
6
7 kernel<<<GRID,BLOCK>>>(SIZE, uvm_x);
8
9 cudaDeviceSynchronize();

```

これら 3 つの課題に対処するために NVIDIA は、ホスト側とデバイス側のどちらからもアクセス可能な単一の仮想アドレス空間を扱えるようにする UVM (Unified Virtual Memory) を提供している [14]。CUDA 8.0 以降では、cudaMallocManaged 関数を用いてメモリ領域を確保することにより、単一のポインタでホスト側、およびデバイス側の両方からアクセスすることができる。UVM を使用して記述された GPU プログラムのホストコードであるソースコード 4 を例に挙げると、UVM を使用することにより、メモリ管理に必要なポインタが uvm_x の 1 つに

集約されていることが確認できる。また、ソースコード 3 の 7 行目、および 15 行目に記述されているホスト・デバイス間のデータ転送を行う関数を省略できていることも確認できる。

3.2 ページフォルト処理概要

UVM を利用することでプログラミングの抽象度が高まるという利益がある一方、問題点も存在する。それが UVM を用いた GPU プログラムでデータにアクセスする際に発生する可能性があるページフォルトである。ページフォルトとはデバイスのメモリアクセス時に物理メモリ上にページ割り当てがないと発生する例外である。デバイス側のメモリアクセスでページフォルトが発生した場合、20 μ s 以上を要するページフォルト処理 (PageFault Handling) が行われる [15]。ページフォルト処理中は大きなレイテンシを伴う処理であるため SM がストールする可能性が高く、GPU のスループットを著しく低下させるオーバヘッドとなる。そこで、NVIDIA や AMD の GPU では、ページフォルト処理のオーバヘッドを抑制するために、複数のページフォルトを集約して処理する実行モデルが採用されている [8]。ページフォルト処理はハードウェアの進化により部分的な改善はされているものの根本的な解決には至っていない。ページデータの転送時間は PCIe (Peripheral Component Interconnect Express) の世代交代により大幅に短くなっている。PCIe Gen 3.0 では $\times 16$ 接続において帯域幅が 15.75 GB/s であった。次の世代の PCIe Gen 4 では $\times 16$ 接続において帯域幅が 31.5 GB/s と帯域幅が 2 倍になった。しかし、ページフォルト処理は大部分を OS や UVM ドライバなどのソフトウェア処理によるレイテンシが占めている。RTX2060 の場合ベースクロックが 1365 MHz であり、この場合 20 μ s は 27300 サイクルに相当するためページフォルト処理により 1 回につき 27300 サイクル以上のレイテンシを伴う。RTX5090 はベースクロックが 2017 MHz であり、この場合 20 μ s は 40340 サイクルに相当するためページフォルト処理 1 回につき 40340 サイクル以上のレイテンシを伴う。このように、GPU が高クロックになるほどページフォルトによるレイテンシの影響は大きくなる。

ここで、図 6 を用いて、この実行モデルにおけるページフォルト発生時の動作を説明する。この例では動作している SM は 3 つとし、それぞれの SM には Warp が 1 つのみ割り当てられているとする。ページフォルトは GPU メモリ管理ユニット (GMMU) によって検知され、その後ページフォルト処理が行われる。このページフォルト処理は 2 つのステップからなっている。ステップ 1 は、GPU からホスト OS へ割り込み処理を行い UVM ドライバにページフォルトの発生を通知し、ページフォルトバッファから

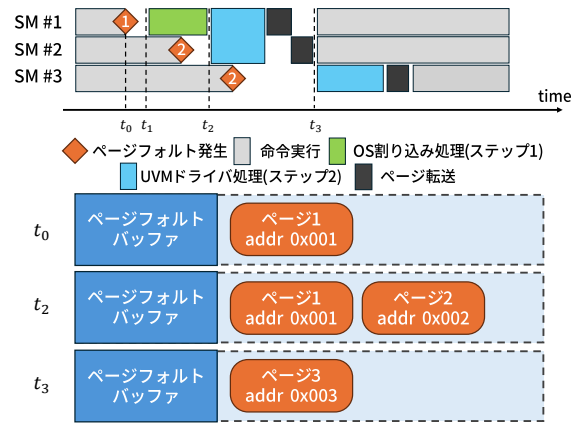


図 6: ページフォルト処理動作例

ページフォルトの情報の読み出しを行う。ステップ 2 は、UVM ドライバにより必要ページデータの転送準備を行う。ステップ 1 の処理中は後続で発生したページフォルト処理をまとめて集約することが可能である。まず、時刻 t_0 で GPU メモリ管理ユニット (GMMU) がページ 1 に対するページフォルトを検知すると、ページ 1 のページフォルトに関する情報をページフォルトバッファに追加する。その後 GMMU はページフォルトバッファにページ情報が追加されたことを UVM ドライバに通知し、ページフォルト処理のステップ 1 を開始する (t_1)。このステップ 1 が完了するまでに発生した後続のページフォルトは集約されて 1 つのページフォルト処理として行われる。そのため図 6 の例ではページ 1、ページ 2 の 2 つが集約されたページフォルト処理が行われる。このように、ページフォルトバッファに複数のエントリが存在する場合、複数のページフォルトを集約しページフォルト処理のオーバヘッドを抑制している。ページフォルト処理のステップ 1 が完了すると、ステップ 2 の処理が開始される。ステップ 2 の処理開始からページ転送が完了するまでの間 (図 6 における $t_2 \sim t_3$)、新たに発生したページフォルトによる一連の処理を開始することができない。一方で、ページフォルトに関する情報はページフォルトバッファに随時追加されるため、 t_2 以降では、ページ 3 の情報がページフォルトバッファに存在している。ページ 1、ページ 2 の転送が完了した t_3 において、ページ 3 のページフォルト処理が開始される。

4. データ依存関係に考慮した PTX の命令順序の提案

本章では、どのような PTX 命令順序が良いかを論じ、その上でそのように命令順序を変更する具体的なアルゴリズムを示す。提案手法では、ロード命令を連続的に配置しページフォルト処理を可能な限り集約する。そして、ページフォルトや TLB ミスを引き起こしたロード命令とデータ依存関係がない命令を先に実行することで、ページフォ

ルト処理時間や TLB ミスの処理時間を有効活用する。

4.1 PTX 命令順序変更方針

UVM を利用した際に発生するページフォルト処理はハードウェアの進化により改善された部分もあるが、依然として大きなオーバーヘッドとなっている。ページフォルト処理中のページデータ転送時間に関しては PCIe の進化により高速化されている。しかし、ページフォルト処理では大部分の時間がソフトウェアによる処理であるため未解決である。また、GPU が高クロックであるほどページフォルト処理によるオーバーヘッドの影響は大きくなる。この問題を解決するため、本研究では GPU が実行する命令順序に焦点を向ける。市販の GPU でも利用可能なソフトウェアからのアプローチとして、ページフォルト処理の集約およびページフォルト処理時間を有効活用できる命令順序を構成するアルゴリズムを提案する。

本研究では、汎用性を持たせるため、命令順序の変更を考えるうえで PTX を解析の対象として命令順序の変更を行った。CUDA では仮想 ISA として **Parallel Thread Execution (PTX)** を持ち、基本的にハードウェアに依存しない。一方でネイティブ ISA は **Streaming Assembly (SASS)** であるが、これはハードウェア依存であり GPU のアーキテクチャごとに異なっている。それがゆえ、本研究では SASS を直接操作することはせず、PTX レベルでの解析・変更を行う。

ここで NVCC コンパイラにより PTX に変換されたソースコード 5 を用いて NVCC コンパイラが生成する PTX をそのまま使うと発生する問題点について説明する。問題点の 1 つ目は、データ依存関係を考慮して最適化されていない点である。ロード命令 (ソースコード 5: 5 行目) が書き込むレジスタ `%f1` を (ソースコード 5: 7 行目) ですぐに利用している。つまり、ロード命令でページフォルトが発生している場合依存関係のある後続命令 (ソースコード 5: 7 行目) でストールしてしまう。ロード命令でページフォルトが発生しても、そのデータに依存しない命令が続く限り、それらは実行可能である。11 ~ 14 行目では 5 行目とデータ依存関係は存在しない。7 行目より前に 11 ~ 14 行目の命令を実行した場合ストールは発生しないため、これらの命令は前倒しして実行することが望ましい。ロード命令とデータ依存関係のある命令の実行前に、ロード命令とデータ依存関係のない命令を実行する。これにより、ページフォルト処理のオーバーヘッドや TLB ミスのオーバーヘッドの一部を隠蔽することができる。

問題点の 2 つ目は、ロード命令が連続して配置されず、分散してしまっていることである。3 章で述べたようにページフォルト処理は大きく分けて 2 つのステップからなっている。最初のページフォルト発生による OS への割り込み

処理、そしてページフォルトバッファからの読み出しが行われるまでに発生した後続のページフォルト処理は同一のバッチとして集約処理される。つまり、ページフォルトが発生する可能性がある命令をできるだけ連続させることで 1 回のバッチで複数のページフォルト処理をまとめることが可能である。また、ロード命令とデータ依存関係のある命令よりも前でロード命令を連続させることで SM がストールし後続のロード命令が発行できないことを防げる。ページフォルト処理は少なくとも数十 μ s を要し、ns レベルで稼働している GPU にとっては大きなオーバーヘッドである。そのため、ページフォルトを集約できるか否かが性能を大きく左右する。これらから、ロード命令はできるだけ早く、そして連続して発行することが望まれる。

ソースコード 5: NVCC コンパイラにより生成された PTX

```

1 //省略
2 cvta.to.global.u64      %rd8, %rd4;
3 mul.wide.s32            %rd9, %r1, 4;
4 add.s64                 %rd2, %rd8, %rd9;
5 ld.global.f32           %f1, [%rd2+4];
6
7 cvt.f64.f32             %fd1, %f1;      //ロード命令の
8                               結果に依存
9 mul.f64                 %fd2, %fd1, 0d3FD3333333333333;
10
11 cvta.to.global.u64      %rd10, %rd5;
12 mul.wide.s32            %rd11, %r12, 4;
13 add.s64                 %rd12, %rd10, %rd11;
14 ld.global.f32           %f2, [%rd12+4];
15
16 cvt.f64.f32             %fd3, %f2;      //ロード命令の
17                               結果に依存
18 cvta.to.global.u64      %rd3, %rd7;
19 mul.wide.s32            %rd13, %r10, 4;
20 add.s64                 %rd14, %rd3, %rd13;
21 ld.global.f32           %f3, [%rd14+8];
22
23 cvt.f64.f32             %fd4, %f3;      //ロード命令の
24                               結果に依存
25 mul.f64                 %fd5, %fd4, 0d3FD3333333333333;
26 fma.rn.f64             %fd6, %fd2, %fd3, %fd5;
27
28 add.s64                 %rd15, %rd1, %rd13;
29 ld.global.f32           %f4, [%rd15+8];
30
31 cvt.f64.f32             %fd7, %f4;      //ロード命令の
32                               結果に依存
33 add.f64                 %fd8, %fd6, %fd7;
34 cvt.rn.f32.f64          %f5, %fd8;
35 st.global.f32           [%rd15+8], %f5;
36
37 ld.global.f32           %f6, [%rd2+4];
38 cvt.f64.f32             %fd9, %f6;
39 mul.f64                 %fd10, %fd9, 0d3FD3333333333333;
40 ld.global.f32           %f7, [%rd12+4];
41 cvt.f64.f32             %fd11, %f7;
42 ld.global.f32           %f8, [%rd14+8];
43 //省略

```

これを踏まえて PTX を確認すると、ソースコード 5 は問題点 1 で指摘したように、データ依存関係で後続命令が発行できないためページフォルトが発生した場合同一 Warp 上ではソースコード 5 の 5 行目, 14 行目, 21 行目, 29 行目のロード命令が個別のページフォルト処理として発行される可能性が高く、効率が悪いことが確認できる。また、問題点 2 で指摘したようにソースコード 5 の 37 行目, 42 行目のロード命令はロードするアドレス計算は事前に終わっているが連続して配置されておらず分散してしまっている。これらのロード命令はアドレス計算が終わった段階で発行できるため、より早い段階で発行可能な命令である。

ソースコード 6: データ依存関係を考慮した PTX

```
1 //省略
2 cvta.to.global.u64    %rd8, %rd4;
3 mul.wide.s32          %rd9, %r1, 4;
4 add.s64                %rd2, %rd8, %rd9;
5 ld.global.f32         %f1, [%rd2+4];
6 ld.global.f32         %f6, [%rd2+4];
7
8 cvta.to.global.u64    %rd10, %rd5;
9 mul.wide.s32          %rd11, %r12, 4;
10 add.s64               %rd12, %rd10, %rd11;
11 ld.global.f32         %f2, [%rd12+4];
12 ld.global.f32         %f7, [%rd12+4];
13
14 cvta.to.global.u64    %rd3, %rd7;
15 mul.wide.s32          %rd13, %r10, 4;
16 add.s64               %rd14, %rd3, %rd13;
17 ld.global.f32         %f3, [%rd14+8];
18 ld.global.f32         %f8, [%rd14+8];
19
20 add.s64               %rd15, %rd1, %rd13;
21 ld.global.f32         %f4, [%rd15+8];
22
23 cvt.f64.f32           %fd1, %f1;  //ロード命令の
24                           結果に依存
25 mul.f64               %fd2, %fd1, 0d3FD3333333333333;
26 cvt.f64.f32           %fd3, %f2;
27 cvt.f64.f32           %fd4, %f3;
28 mul.f64               %fd5, %fd4, 0d3FD3333333333333;
29 fma.rn.f64            %fd6, %fd2, %fd3, %fd5;
30 cvt.f64.f32           %fd7, %f4;
31 add.f64               %fd8, %fd6, %fd7;
32 cvt.rn.f32.f64        %f5, %fd8;
33 st.global.f32         [%rd15+8], %f5;
34 cvt.f64.f32           %fd9, %f6;
35 mul.f64               %fd10, %fd9, 0d3FD3333333333333;
36 cvt.f64.f32           %fd11, %f7;
37 //省略
```

これら 2 つの問題点を解消し、データ依存関係を考慮して最適化した PTX をソースコード 6 に示す。問題点 1 で指摘したロード命令に依存する命令が直後に配置されることによる非効率性は、データ依存関係にない命令を前方に移動することで可能な限りロード命令に依存する命令の発行が遅くなるように命令順序を変更することで解決できる。また、問題点 2 で指摘したロード命令が分散配置され

ている問題に対しては、データ依存関係を考慮に入れたうえでロード命令を可能な限り連続発行できるように命令順序を変更することで解決できる。

ソースコード 7: 命令順序変更を複雑にする PTX 構文例

```
1 .....//省略
2
3     mov.u32            %r15, %tid.x;
4     add.s32            %r1, %r14, %r15;
5     setp.ge.s32        %p1, %r1, %r12;
6     @%p1 bra           $L_BB0_4;
7
8     cvta.to.global.u64 %rd18, %rd14;
9     cvt.s64.s32        %rd2, %r1;
10    add.s64             %rd3, %rd18, %rd2;
11    ld.global.u8        %rs1, [%rd3];
12    setp.eq.s16         %p2, %rs1, 0;
13    @%p2 bra           $L_BB0_7;
14
15 $L_BB0_4:
16    ld.global.s32       %rd10, [%rd29];
17    add.s64             %rd25, %rd7, %rd10;
18    ld.global.u8        %rs3, [%rd25];
19    setp.ne.s16         %p4, %rs3, 0;
20    @%p4 bra           $L_BB0_6;
21
22    ld.global.u32       %r16, [%rd5];
23    add.s32             %r17, %r16, 1;
24    shl.b64             %rd26, %rd10, 2;
25    bar.sync
26    .....//省略
27
28 $L_BB0_6:
29    add.s64             %rd29, %rd29, 4;
30    add.s32             %r18, %r22, %r23;
31    add.s32             %r21, %r21, 1;
32    setp.lt.s32         %p5, %r21, %r18;
33    @%p5 bra           $L_BB0_4;
34
35 $L_BB0_7:
36    ret;
37
38 .....//省略
```

4.2 提案手法の設計

PTX の命令順序を変更するにあたり制御依存、データ依存を遵守する必要がある。また、PTX ではスレッドブロック内の全スレッドを同期させるバリア命令も存在するため注意する必要がある。そのうえで、ページフォルトを集約できるようにデータ依存関係を考慮したロード命令の先送りやロード命令の連続した発行ができる PTX 命令順序を構築する必要がある。

まず、制御依存、バリア命令への対処を説明する。説明用の PTX としてソースコード 7 を用いる。制御命令として PTX にはソースコード 7 の 6 行目, 13 行目にあるような分岐命令が存在し、分岐先ラベルとしてソースコード 7 の 15 行目, 28 行目にあるようなラベルがある。そして、

アルゴリズム 1 PTX の命令順序を決定するアルゴリズム

Require: Dependency Graph G , ▷ 依存グラフ
In-degree array D ▷ 入次数
Ensure: Result Order R ▷ 最適化命令順序の出力先

```

1: PriorityQueue  $pq$  ▷ プライオリティキュー
2:
3: ▷ 入次数 0 のノードをキューに追加
4: for each Instruction  $ins$  in Original PTX do
5:   if  $D[ins] == 0$  then
6:      $pq.push((is\_ld, use\_ld\_result, line\_number), ins)$ 
7:   ▷ 優先度はタプルの辞書順
8:   end if
9: end for
10:
11: ▷ プライオリティキューが空になるまでループ
12: while  $pq$  is not empty do
13:    $u \leftarrow pq.pop()$  ▷ 優先度が一番高いものを取り出す
14:   append  $u$  to  $R$  ▷ 最適化命令順序に追加
15:
16:   ▷ 最適化命令順序に追加した命令により
   入次数が変化する命令の処理
17:   for each  $v \in G[u]$  do
18:      $D[v] \leftarrow D[v] - 1$  ▷ 入次数をデクリメント
19:
20:     ▷ 入次数が 0 ならプライオリティキューに追加
21:     if  $D[v] == 0$  then
22:        $pq.push(v)$ 
23:     end if
24:   end for
25: end while

```

バリア命令としてソースコード 7 の 25 行目にあるような命令が存在する。制御依存、ラベル、バリア命令を超えて命令を移動することはプログラムの意味を変える可能性がある。制御命令、ラベル、バリア命令で挟まれた区間を 1 つのブロックとして捉え、このブロックを命令ブロックと呼ぶ。そして、命令ブロックを跨ぐ命令順序の変更を禁止し、命令ブロック内でのみ命令順序の変更を許可する。これにより、プログラムの意味が変わらないことを担保する。また、データ依存に関しては PTX 全体を通して命令で利用しているレジスタを記録し依存グラフを作成する。そして、命令ブロックを識別するために各命令より後に存在する制御命令、ラベル、バリア命令のいずれかに対しても依存グラフを作成する。これらの処理を PTX に対して行ったあと、アルゴリズム 1 に示す疑似コードに従い命令順序を変更する。

アルゴリズム 1 の Require の入次数は依存グラフを作成する際の各命令が発行可能になるまでの命令数 (入次数) を示している。命令順序の優先順位の決定にはプライオリティキューを使用する。プライオリティキューの優先度を決定する条件を次に示す。

- (1) ロード命令であるか (`is_ld`)
- (2) ロード命令の結果に依存しない命令であるか

(`use_ld_result`)

- (3) 優先順位が同率の場合は元の PTX の命令順序に従う (`line_number`)

番号が小さい条件ほど優先順位が高くなるように設定する。ロード命令を最優先命令として定義し、ロード命令の結果に依存する命令は優先度を下げる。意図しない命令順序になることを防ぐため、優先度が同率の場合元の PTX の命令順序に従い優先度を決定する。

プライオリティキューには最初、入次数が 0 の命令群を追加する (アルゴリズム 1: 4 ~ 9 行目)。その後、命令を最適化した命令順序に追加するごとに、その命令の結果とデータ依存がある全命令の入次数をデクリメントしていく (アルゴリズム 1: 18 行目)。この処理により入次数が 0 になった命令はプライオリティキューに追加する (アルゴリズム 1: 21 ~ 23 行目)。

5. 評価と考察

本章では、4 章で述べた提案手法を用いて生成した PTX と、NVCC コンパイラが生成した PTX を実行比較し、提案手法の有効性を検証する。

表 1: 実行環境

OS	Ubuntu 22.04.5
CUDA	11.7
CPU	Intel Core i9-7900X
clock	3.3 GHz
GPU	TITAN RTX
SM	72 units
CUDA Core	4,508 units

表 2: 評価に用いたベンチマークと入力パラメータ

Rodinia-3.1			
backprop			262144
bfs			graph4194304.txt
b+tree	file mil.txt	command	command.txt
hotspot	1024 2 500 temp_1024	power_1024	
kmeans			204800.txt

5.1 評価環境

評価環境を表 1 に示す。CPU は Intel Core i9-7900X を使用し、GPU は Turing アーキテクチャを採用した TITAN RTX を使用して計測を行った。ベンチマークプログラムには、Rodinia-3.1 ベンチマークスイート [16] から b+tree, backprop, bfs, hotspot, kmeans を使用した。使用した入力パラメータを表 2 に示す。各ベンチマークに対して NVCC コンパイラの -O3 オプションで作成した PTX と提案手法を適用した PTX をそれぞれ実行し比較

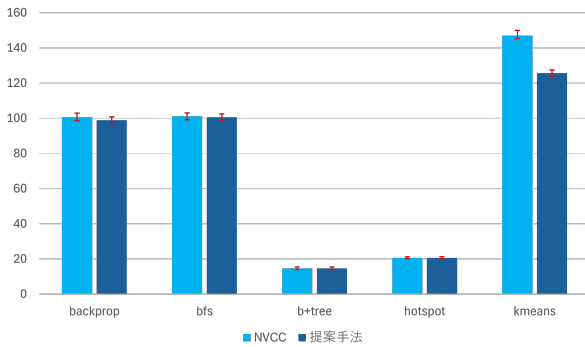


図 7: ページフォルト処理回数の評価

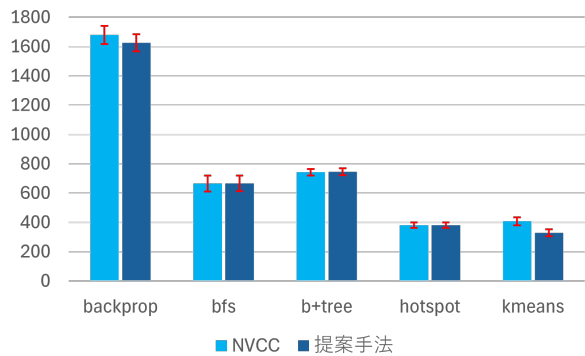


図 8: ページフォルト発生回数

した．全ての結果は 1000 回実行した平均値および標準偏差を示した．

5.2 ページフォルト処理回数

ページフォルト処理回数の評価結果を図 7 に示す．backprop では約 1.7%，bfs では約 0.6%，kmeans では約 14.5% ページフォルト処理回数が削減された．一方，b+tree と hotspot ではページフォルト処理回数に変化が見られなかった．

5.3 ページフォルト発生回数

ページフォルト発生回数の評価を図 8 と図 9 に示す．図 8 はカーネル関数全体で発生したページフォルト数，図 9 はカーネル関数別で発生したページフォルト数を示す．カーネル関数全体では backprop では約 3.2%，kmeans では約 19.3% ページフォルト発生回数が削減できた．その他のベンチマークでは差がほとんど見られなかった．カーネル関数別では backprop の kernel2 では約 6.3%，kmeans の kernel1 では約 37% ページフォルト発生数が削減できた．一方，b+tree の kernel2 では約 0.9%，kmeans の kernel2 では約 1.1% ページフォルト発生数が増加した．その他のカーネルではフォルト発生回数に変化が見られなかった．

5.4 カーネル関数全体実行時間

カーネル関数全体実行時間の評価結果を図 10 に示す．

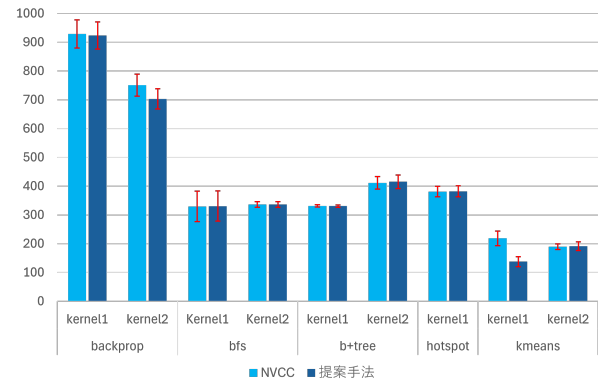


図 9: カーネル関数別ページフォルト発生回数

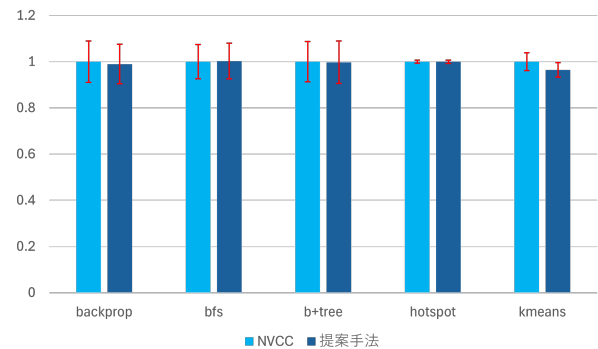


図 10: 正規化したカーネル関数全体実行時間比

図 10 は NVCC コンパイラで作成した PTX のカーネル関数全体実行時間で正規化したものである．backprop では約 1.0%，kmeans では約 3.0%，それぞれ実行時間が削減された．一方，bfs，b+tree，hotspot では実行時間に変化は見られなかった．

5.5 考察

backprop，bfs，kmeans ではページフォルト処理回数が削減された．これは提案手法が有効に機能した結果，ロード命令を連続的に実行し，そしてロード命令の結果を利用する命令を後ろ倒ししストールする回数を削減できたことを示している．ストールする回数を削減できたことでロード命令の発行を連続させることが可能となった．また，backprop と kmeans ではページフォルト発生回数も削減された．先行して発行できたロード命令により発生したページフォルトのプリフェッチで転送されたページデータが有効に利用された．そのため，ページフォルトの発生回数を削減できたと考える．ページフォルト処理回数そしてページフォルト発生回数が削減されたことにより，カーネル全体実行時間を削減することができた．

ページフォルト処理を集約させることはカーネル関数の複数回の呼び出しの間で行うことはできない．そのため，提案手法の影響の大きさを確認するために各ベンチマークのカーネル関数実行ごとのページフォルト処理回数およ

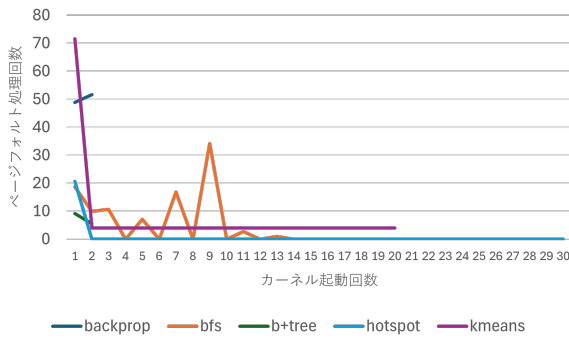


図 11: カーネル関数実行ごとのページフォルト処理回数 (NVCC)

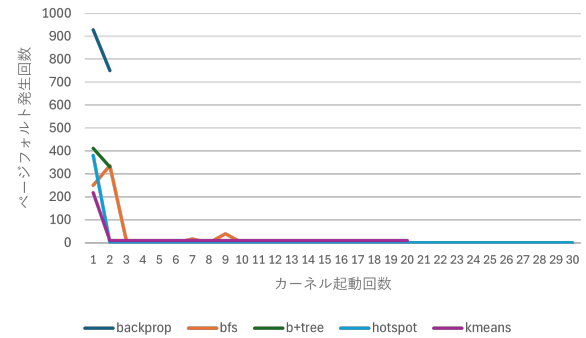


図 13: カーネル関数実行ごとのページフォルト発生回数 (NVCC)

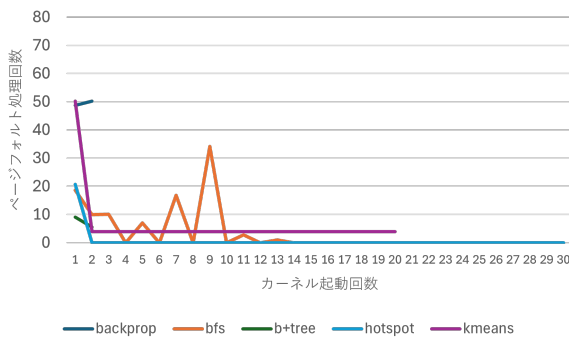


図 12: カーネル関数実行ごとのページフォルト処理回数 (提案手法)

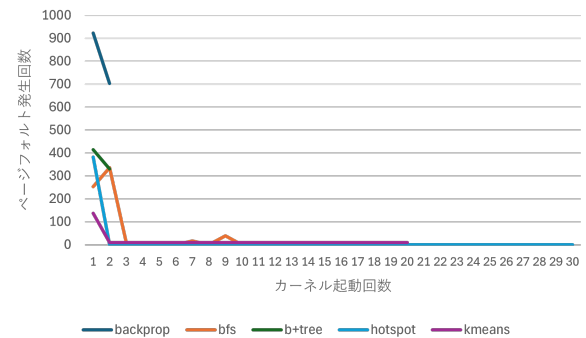


図 14: カーネル関数実行ごとのページフォルト発生回数 (提案手法)

びページフォルト発生回数を計測した。各ベンチマークのカーネル関数実行ごとでページフォルト処理が行われた回数を図 11 と図 12 にそれぞれ示す。そして、各ベンチマークのカーネル関数実行ごとのページフォルト発生回数を図 13 と図 14 にそれぞれ示す。なお、hotspot は 2 回目以降のカーネル関数の実行ではページフォルト処理およびページフォルトの発生はなかったためグラフでは 31 回目以降を省略してある。

提案手法ではロード命令を連続的に配置しページフォルト処理回数を削減するように PTX の命令順序を変更した。そのため、提案手法の影響が大きいものはベンチマーク特性としてページフォルト処理回数、ページフォルト発生回数が比較的多いと予想される。backprop はページフォルト処理、ページフォルト発生が他のベンチマークと比較して多数発生しているが、これは提案手法の効果が大きかったことと整合する。bfs はページフォルト処理回数は backprop と同等であるが提案手法の効果は小さかった。bfs はカーネル関数あたりのページフォルト発生回数は比較的少数である。また、2 回目のカーネル関数の実行ではページフォルト発生数は bfs 内で最大であるがページフォルト処理回数は少数である。これは NVCC コンパイラで生成された PTX の命令順序ですでにページフォルトの発生が連続的であり、1 回のページフォルト処理に集約して処

理されているためである。そのため、提案手法でロード命令を連続的に配置しても効果があまり得られなかったと考える。また、kmeans はページフォルト発生回数は他のベンチマークと比較して少数であるが、多数のページフォルト処理に分けられていることが確認できる。そのため、提案手法によりロード命令を連続的に発行することでページフォルト処理回数を削減できた。一方、b+tree、hotspot は提案手法の効果が得られなかった。b+tree と hotspot は複数のカーネル関数の実行でページフォルトが発生しなかったため提案手法の効果が得られなかったと考える。

次に、提案手法によってページフォルト発生回数が増加してしまった b+tree の kernel2 と kmeans の kernel2 の原因を究明する。

b+tree の kernel2 に対してコンパイラが生成した PTX の一部分をソースコード 8 に示す。ソースコード 8 の 10 行目にある命令は先行するロード命令とデータ依存関係が存在するため後ろ倒しの対象となる。しかし、10 行目の命令は後続のロード命令ともデータ依存関係がある。ソースコード 9 に提案手法を適用したあとの PTX を示す。先程ソースコード 8 の 10 行目にあった命令は先行するロード命令とデータ依存関係があったため後ろ倒しされ、ソースコード 9 の 19 行目に移動している。それに伴いデータ依存のある後続の命令も同様に後ろ倒しされた結果、ソー

スコード 9 の 19 行目とデータ依存のある後続のロード命令 (ソースコード 9: 25 行目) も後ろ倒しの対象となってしまった。早く発行すべきロード命令もまとめて後ろ倒しの対象に選ばれたため、提案手法を適用したことでページフォルト発生回数が増加してしまったと考える。これに対して、提案手法で後ろ倒しされる命令内でロード命令が存在する場合優先的に発行できるように命令順序を変更することで、ロード命令が後ろ倒しされることを緩和できると考える。

kmeans の kernel2 はページフォルトが発生する可能性のある命令が 1 命令しか存在しない。命令順序の変更によりレジスタプレッシャーが高まり、性能低下の要因となるレジスタスピルが発生する懸念があったため、ローカルメモリとの通信を確認した。その結果、すべてのベンチマークにおいて NVCC が生成した PTX と提案手法を適用した PTX の両者の実行でローカルメモリとの通信は確認されなかった。つまり、レジスタスピルがページフォルトの発生数を増加した原因ではない。一方、ロード命令の発行タイミングは提案手法により NVCC コンパイラの PTX とは発行タイミングが異なっている。GPU は複数の SM が並列で計算しておりロード命令の発行タイミングのずれが最終的にページフォルト発生数の増加に影響を与えた可能性や測定誤差の可能性などが考えられる。

ソースコード 8: コンパイラが生成した PTX

```
1 mov.u32      %r5, %ctaid.x;
2 cvt.s64.s32  %rd1, %r5;
3 cvta.to.global.u64 %rd65, %rd61;
4 mul.wide.s32 %rd66, %r5, 8;
5 add.s64      %rd2, %rd65, %rd66;
6 mov.u32      %r6, %tid.x;
7 cvt.s64.s32  %rd3, %r6;
8 ld.global.u64 %rd150, [%rd2];
9 cvta.to.global.u64 %rd5, %rd60;
10 mul.lo.s64   %rd67, %rd150, 2068;
11 //先行するロード命令の結果に依存
12 (後ろ倒しの対象)
13
14 add.s64      %rd68, %rd5, %rd67;
15 mul.wide.s32 %rd69, %r6, 4;
16 add.s64      %rd70, %rd68, %rd69;
17 ld.global.s32 %rd153, [%rd70+1032];
18 cvta.to.global.u64 %rd71, %rd63;
19 add.s64      %rd7, %rd71, %rd66;
20 ld.global.u64 %rd151, [%rd7];
21 cvta.to.global.u64 %rd72, %rd62;
22 add.s64      %rd9, %rd72, %rd66;
23 cvta.to.global.u64 %rd73, %rd64;
24 add.s64      %rd10, %rd73, %rd66;
25 setp.lt.s64  %pl, %rd54, 1;
```

実験結果をまとめると、一部のベンチマークに対しては大きな効果があり、効果がないまたはページフォルトの増加があるベンチマークに対しても性能の変化は非常に限定的である。これらより、提案手法はページフォルト処理の

集約に対して有効であると結論付ける。ただし、ロード命令が後ろ倒しされすぎ、意図せずページフォルトが増加する問題を解決することは今後の課題となる。今後の展望としては、さらに高度な依存解析を用いた命令ブロックをまたぐ命令順序変更による、さらなる性能改善が挙げられる。

ソースコード 9: 提案手法を適用した PTX

```
1 mov.u32      %r5, %ctaid.x;
2 cvt.s64.s32  %rd1, %r5;
3 cvta.to.global.u64 %rd65, %rd61;
4 mul.wide.s32 %rd66, %r5, 8;
5 add.s64      %rd2, %rd65, %rd66;
6 ld.global.u64 %rd150, [%rd2];
7 mov.u32      %r6, %tid.x;
8 cvt.s64.s32  %rd3, %r6;
9 cvta.to.global.u64 %rd5, %rd60;
10 mul.wide.s32 %rd69, %r6, 4;
11 cvta.to.global.u64 %rd71, %rd63;
12 add.s64      %rd7, %rd71, %rd66;
13 ld.global.u64 %rd151, [%rd7];
14 cvta.to.global.u64 %rd72, %rd62;
15 add.s64      %rd9, %rd72, %rd66;
16 cvta.to.global.u64 %rd73, %rd64;
17 add.s64      %rd10, %rd73, %rd66;
18 setp.lt.s64  %pl, %rd54, 1;
19 mul.lo.s64   %rd67, %rd150, 2068;
20 //先行するロード命令の結果に依存
21 //後続のロード命令とも間接的に依存関係
22
23 add.s64      %rd68, %rd5, %rd67;
24 add.s64      %rd70, %rd68, %rd69;
25 ld.global.s32 %rd153, [%rd70+1032];
26 //ロード命令が後ろ倒しされている
```

表 3: 使用したベンチマーク

PolyBench/GPU 1.0	
GEMM	General Matrix-Matrix Multiplication
The SHOC Benchmark Suite	
SPMV	Sparse Matrix-Vector Multiplication (csr_scalar)

5.6 PTX 命令順序変更による性能向上の限界に関する調査

提案手法による GPU 性能向上の限界を考察するにあたり、表 3 に示す算術演算ベンチマークを使用した [17, 18]. 図 15 に正規化したカーネル関数実行時間を示す。また、Nsight Compute を用いて測定した計算強度を表 4 に示す。GEMM では実行時間が約 34.4%削減された。SPVM では実行時間に変化は見られなかった。ただし、GEMM では平均してページフォルト処理回数が増加しており、原因は究明できていない。

計算強度が高いベンチマークでは提案手法を適用することで、実行時間を大幅に改善させることが可能である。一方、計算強度が低いものは提案手法を適用しても性能向上は見られない。提案手法ではロード命令の集約、前倒し、

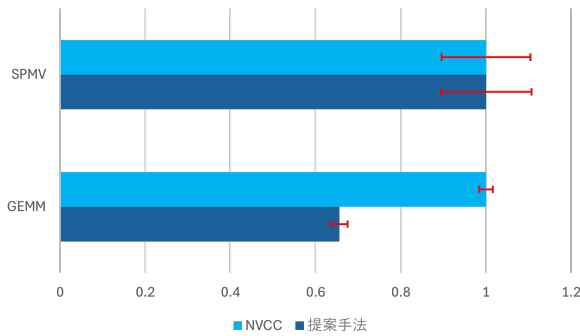


図 15: 正規化したカーネル関数実行時間比

そしてロード命令とデータ依存関係がある命令の後ろ倒しを行った。計算強度が高いベンチマークでは、PTX 命令順序を最適化することで複数のロード命令を連続的にすることが可能である。これにより、ページフォルトの集約やロード命令とデータ依存が存在しない命令でページフォルト処理のオーバーヘッドを一部隠蔽可能である。しかし、計算強度が低いベンチマークではメモリアクセス命令に対して計算命令の割合が少数である。また、間接メモリアクセスなどにより後続の命令を先に計算することが困難である。これらから、PTX 命令順序を最適化しロード命令を連続的にすることや、ページフォルト処理のオーバーヘッドを隠蔽することが不可能である。

提案手法は計算強度が高いベンチマークでより効果的に PTX 命令順序を最適化することが可能である。一方、間接メモリアクセスを含む計算強度の低いベンチマークでは、PTX 命令順序の最適化のみによる性能向上には、限界が存在すると結論づける。

表 4: 演算強度 [FLOP/byte]

	NVCC	提案手法
GEMM	6.16	5.97
SPMV	0.25	0.25

6. おわりに

近年の GPU は Unified Virtual Memory (UVM) と呼ばれる機能をサポートしており、GPU プログラミングにおけるメモリ管理の抽象度を高めている。その一方で、デバイス側のメモリアクセスでページフォルトが発生した場合、20 μ s 以上を要するページフォルト処理がオーバーヘッドとなりうる。このオーバーヘッドを抑制するために、GPU では複数のページフォルト処理を集約して実行する設計がされている。また、先行研究では追加ハードウェアを利用しこのオーバーヘッド中に、SM に割り当てられているスレッドブロックを切り替えページフォルトを追加発生させる手法が提案されている。しかし、多くの既存手法は GPU 内に

ハードウェア回路の追加実装を必要とする。市販の GPU での実行を高速化するためには、ソフトウェアからのアプローチが必要となる。

そこで本論文では、現状の NVCC コンパイラでは制御できていないデータ依存関係に考慮した PTX の命令順序そして、命令順序をチューニングする設計を提案した。提案手法ではデータ依存関係を考慮しロード命令を可能な限り前倒しし、ロード命令を連続的に配置しページフォルト処理を可能な限り集約する。また、データ依存関係がある命令は後ろ倒しすることでロード命令とデータ依存関係がある命令でストールすることを可能な限り遅くする。そして、ロード命令をストールする前に連続的に発行可能にする。

提案手法の有効性を検証するため、TITAN RTX を搭載した実行環境で Rodinia-3.1 を使用して評価を行った。提案手法を適用した結果 NVCC コンパイラが作成した PTX と比較して、提案手法を適用した PTX ではページフォルト処理回数では最大 14.5%、ページフォルト発生回数では最大 19.3% の削減を達成した。一方で、提案手法を適用することでページフォルト発生回数が増加したベンチマークも存在した。これに関して、提案手法を適用した PTX からの課題を考察した。今後は、本手法でロード命令が後ろ倒しされる問題を改善することで意図しないページフォルトの増加を抑制する。そして、本手法では命令順序を変更できていない、命令ブロックをまたいだ命令順序の変更をすることで、性能改善が得られなかったベンチマークの性能向上を目指す。

参考文献

- [1] Keckler, S. W., Dally, W. J., Khailany, B., Garland, M. and Glasco, D.: GPUs and the Future of Parallel Computing, *IEEE Micro*, Vol. 31, No. 5, pp. 7–17 (2011).
- [2] Mostak, T.: Map-D: World's Fastest Database and Big Data Analytics Platform, *GPU Technology Conference (GTC)* (2014).
- [3] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T.: Caffe: Convolutional Architecture for Fast Feature Embedding, *Proceedings of the 22nd ACM International Conference on Multimedia* (2014).
- [4] Top500: Top500 List, <http://www.top500.org>. (Accessed 2026-02-01).
- [5] 松森健明: クラウド HPC の最先端: 未来を切り拓くスーパーコンピュータ Microsoft Azure Eagle, https://www.pccluster.org/ja/event/data/240627_pccc_wsSuzukakedai_27-11_matsumori.pdf (2024). (Accessed 2026-02-01).
- [6] NVIDIA Corporation: NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU (2016).
- [7] The Linux Kernel Organization: Heterogeneous Memory Management (HMM), [url: https://www.kernel.org/doc/html/latest/mm/hmm.html](https://www.kernel.org/doc/html/latest/mm/hmm.html).

- (Accessed 2026-02-01).
- [8] Kim, H., Sim, J., Gera, P., Hadidi, R. and Kim, H.: Batch-Aware Unified Memory Management in GPUs for Irregular Workloads, *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)* (2020).
 - [9] NVIDIA Corporation: CUDA Documentation, <https://docs.nvidia.com/cuda/>. (Accessed 2026-02-01).
 - [10] NVIDIA Corporation: *NVIDIA TESLA V100 GPU ARCHITECTURE, THE WORLD'S MOST ADVANCED DATA CENTER GPU* (2016).
 - [11] Choquette, J., Foley, D. et al.: Volta: Performance and Programmability, *IEEE Micro*, Vol. 38, No. 2, pp. 42–52 (2018).
 - [12] NVIDIA Corporation: CUDA Programming Guide v13.1, <https://docs.nvidia.com/cuda/cuda-programming-guide>. (Accessed 2026-02-01).
 - [13] Da Yan, W. W. and Chu, X.: Optimizing Batched Winograd Convolution on GPUs, *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 20)*, ACM (2020).
 - [14] Sakharaykh, N.: Maximizing Unified Memory Performance in CUDA, <https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/> (2017). (Accessed 2026-02-01).
 - [15] Allen, T. and Ge, R.: Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis, *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021).
 - [16] Che, S. et al.: Rodinia: A Benchmark Suite for Heterogeneous Computing, *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54 (2009).
 - [17] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S. and Cavazos, J.: Auto-tuning a High-Level Language Targeted to GPU Codes, *Proceedings of Innovative Parallel Computing (InPar '12)*, IEEE (2012).
 - [18] 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 (GPGPU-3)*, ACM, pp. 63–74 (2010).