B-038

CUDA プログラミングのためのプロファイリングと最適化支援
Profiling and Optimization support for CUDA programming

大橋　拓也*　　桑原　寛明**　　國枝　義敏**
Takuya Ohashi*　Hiroaki Kuwabara **　Yoshitoshi Kunieda**

1. はじめに

画像描画処理に特化した Graphics Processing Unit (以下, GPU) が一般的な CPU よりも安価で高い性能を発揮するようになり，注目を集めている [1]．GPU の高い処理能力を汎用計算に適用するための技術が，General Purpose computing on GPU(以下, GPGPU)[2] である．NVIDIA 社が提供する Compute Unified Device Architecture(以下，CUDA)[3] は，GPGPU を目的とした開発環境および実行環境である．

CUDA を利用して GPU の性能を引き出すプログラムを作成するためには，GPU の構造や特性を理解する必要がある [4]．しかし，CUDA プログラミングの初心者には GPU の構造や特性などを理解した上で GPU の性能を引き出すプログラムを記述することは困難である．

本稿では CUDA を用いたプログラムの開発と最適化を支援する手法を提案する．提案手法では CUDA プログラムのプロファイリングを行い，性能低下要因を探索して開発者にフィードバックする．プログラムの性能低下要因の一つである，コアレッシングされないグローバルメモリへのアクセスがどこで行われたかを探索し，その原因を調査する．コアレッシングされない原因を示すためにスレッド番号とアドレスの関係図を開発者に示し，コアレッシングされるカーネル関数の修正案を提示することでプログラム開発を支援する．

2. GPGPU 向け開発環境 CUDA

2.1 グラフィックボードの構成

グラフィックボードを構成する要素は，GPU とビデオメモリの 2 つに大別できる．GPU は複数の Streaming Multiprocessor(以下，SM) で構成されており，SM はさらに複数の CUDA コアで構成されている．GPU へのプログラムは CUDA コアが実行する．図 1 にグラフィックボードの内部構成を示す．グラフィックボードに搭載された，ビデオメモリの 1 つであるグローバルメモリは GPU チップの外に置かれており，SM とメモリインタフェースによって接続されている．グローバルメモリは，アクセス速度は低いが，メモリ容量が大きく GPU 上のすべての SM からアクセスできる．

2.2 CUDA プログラミングモデル

CUDA では，CPU をホスト，GPU をデバイスと呼ぶ．デバイス側でホスト側のデータを利用するためには，ホスト側のメモリ領域からデバイス側のメモリ領域へデータをコピーする必要がある．デバイス側のメモリ領域は，ホスト側で明示的に確保しなければならない．データのコピーおよびメモリ領域の確保には，それぞれ CUDA のランタイム API が利用できる．

2.2.1 カーネル関数とスレッド

ホストから呼ばれ，デバイスで実行される関数をカーネル関数と呼ぶ．GPU はカーネル関数をスレッドとして，並列に実行することで高い演算性能を実現する．大量のスレッドを一元管理することは困難であるため，CUDA はグリッドとブロックと

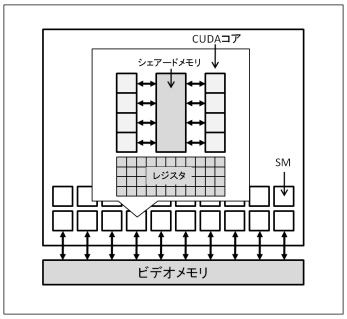

図 1: グラフィックボードの内部構成

いう 2 つの概念で階層的にスレッドを管理する．グリッドは複数のブロックで構成され，ブロックは複数のスレッドで構成される．グリッドとブロックは，それぞれ X 方向，Y 方向，Z 方向の要素を持つ．カーネル関数を実行する各スレッドには個別のスレッド番号が割り振られている．ユーザはスレッド番号により当該スレッドがアクセスする配列の要素を指示できる．CUDA では，1 ブロック内の 32 スレッドをウォープ，ウォープの半分の 16 スレッドをハーフウォープと呼び，カーネル関数をウォープ単位で実行する．

2.2.2 グローバルメモリへのアクセス

グローバルメモリはすべてのスレッドから直接アクセスできるが，アクセス速度は GPU のレジスタなどと比べて 100 倍以上も遅い．グローバルメモリへの読み書きを行うための転送は，32 バイト，64 バイト，128 バイトという大きな単位で行われる．CUDA ではデータ転送を効率よく行うためにコアレッシングという仕組みを利用する．コアレッシングとは，ハーフウォープが同時にメモリアクセスを行うことで，一括してデータ転送を行い，メモリアクセスの効率化を実現する仕組みである．本研究ではハードウェアレベルでの CUDA のサポート範囲であるコンピュートケイパビリティが 1.1 までの GPU を対象とする．コンピュートケイパビリティが 1.1 までの GPU は，以下の条件をすべて満たすことでグローバルメモリへのアクセスがコアレッシングされる．

1. 転送するデータ型サイズが 32，64，128 ビットのいずれかである
2. ハーフウォープの各スレッドがアクセスするアドレスが，スレッド番号の昇順で隣接している
3. 先頭スレッドがアクセスするアドレスが 64 バイト境界にアライメントされている

2.3 CUDA プロファイラ

CUDA を用いて高い性能を発揮するプログラムを記述するには，性能を低下させる要因となっている部分を確認し，適切な最適化手法を用いてプログラミングする必要がある．ユーザがその原因を調べる場合は，原因を調査するための命令を追加したり，何度も実行と修正をしたりしなければならず，ユーザに負担がかかる．CUDA には標準でプログラムの実行をプロファイリングする機能が搭載されており，CUDA プログラムのグローバルメ

*立命館大学大学院理工学研究科
**立命館大学情報理工学部

表 1: 性能低下要因の例

項目名	意味
gld_incoherent	コアレッシングされなかった グローバルメモリからの読み込み
gst_incoherent	コアレッシングされなかった グローバルメモリへの書き込み
divergent_branch	分岐命令でウォープが分断した回数
warp_serialize	バンクコンフリクトの発生回数

モリへのアクセス量などの情報を取得できる．CUDA プロファイラを利用することで，プログラム中にどんな性能低下要因が存在するかを調べることができる．CUDA プロファイラで取得可能な項目のうち性能低下要因となり得るものを表 1 に示す．

3. CUDA プログラムの最適化支援

CUDA のプロファイリング機能では，性能低下要因が具体的にソースコードのどこにあるかを調べることはできない．ユーザが求めるものは，原因の有無ではなく原因が発生している場所と解消方法であるため，CUDA が提供するプロファイリング機能だけではユーザの要求を満たせない．したがって，性能低下原因を調査し，その解消方法を提示する最適化支援が必要である．

本稿では，CUDA プログラムをプロファイリングして性能低下要因を調べ，その原因を解消するプログラムの修正例をユーザに提示するまでの一連の手法を提案する．

3.1 プロファイリングと性能低下要因の探索

性能低下要因を調査するために CUDA プログラムのプロファイリングを行う．グローバルメモリへのアクセスをプロファイリングし，そのアクセスがコアレッシングされるかを調査する．グローバルメモリへのアクセスがコアレッシングされるかを調べるためには，コアレッシング条件と照合するためのデータ型サイズとアクセス先アドレスが必要である．プロファイリングを行うために，`sizeof` 演算子を用いてデータ型サイズを取得する命令，アクセス先配列の添字を記憶してアクセス先アドレスを算出する命令を対象のプログラムに追加する．プロファイリング用の命令を追加したプログラムを実行し，プロファイリング結果を出力する．取得したデータ型サイズとアクセス先アドレスをコアレッシング条件と照らし合わせ，そのグローバルメモリへのアクセスがコアレッシングされたかどうかを出力する．

以下にコアレッシング条件を調べるためのアルゴリズムを示す．まず，転送するデータ型サイズを調べる．取得できるデータ型サイズの単位は `Byte` であるため，4，8，16 のいずれかの場合は次に進み，それ以外の場合はコアレッシングされなかったことを示すメッセージと満たされなかった条件を出力する．次に先頭スレッドのアクセス先アドレスが 64 バイト境界にアライメントされているかを調べる．64 バイト境界にアライメントされていない場合はコアレッシングされなかったことを示すメッセージと満たされなかったコアレッシング条件を出力する．最後にアクセス先アドレスがスレッド番号の昇順に連続しているかを調べる．スレッド番号はアクセス先アドレスの添字を記憶する配列のインデックスである．スレッド番号の昇順にならなければ，連続でないと判断してコアレッシングされないことを示すメッセージと満たされなかった条件を出力する．

3.2 性能低下要因の解消

プロファイリング結果をもとに性能を低下させる原因の解消方法をユーザに提示する．ユーザにコアレッシングされなかった原因を示すために，ユーザが記述したカーネル関数のメモリアクセスを図示する．その上で，コアレッシングされるメモリアクセスを行うカーネル関数の修正例を示す．

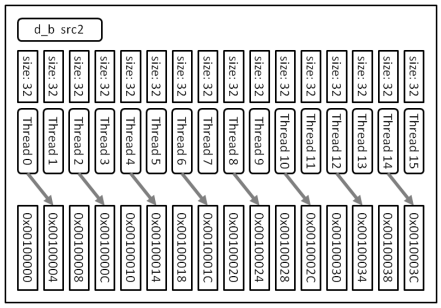

図 2: スレッドとアドレスの関係図の例

```
1 __global__ void
2 kernel1(int *d_a, int *d_b)
3 {
4   int id = threadIdx.x + blockDim.x * blockIdx.x;
5   d_a[id] *= d_b[id + 1];
6 }
```

図 3: 修正前のカーネル関数

図 2 はカーネル関数のメモリアクセスを示す，スレッドとアドレスの関係図の例である．図 2 の 1 段目は転送するデータ型サイズを表し，2 段目はスレッド番号を表す．矢印は当該スレッドがどのアドレスにアクセスしたかを表し，3 段目は当該スレッドのアクセス先アドレスの値を表す．

図 2 は，スレッド番号が偶数のスレッドのみグローバルメモリへアクセスを行っていることに加え，先頭スレッドがアクセスするアドレスが `0x00100004` であるため，64 バイト境界にアライメントされないことを表している．図 2 は，コアレッシング条件の 2 番目と 3 番目を満たさないことにより，コアレッシングされない．

プログラムの修正案として，ユーザが記述したカーネル関数をコアレッシングされるように修正した例を表示する．カーネル関数の修正例は，満たされなかった条件に合わせてアクセスパターンを変えたものを複数表示する．あらかじめ複数のアクセスパターンを用意しておき，ユーザが記述したアクセスパターンと類似するパターンを探して表示する．表示する複数のアクセスパターンは，コアレッシングされるアクセスパターンとなることを保証する．

本研究の目的は，ユーザが何もわからない状態になっている場合に，解決のためのヒントを出す役割を担うことである．本手法で提示するコアレッシングされるアクセスパターンは，ユーザがコアレッシングされるアクセスパターンを学習することを目的とし，本手法が提示したように修正を強制するものではない．1 つの判断基準としてコアレッシングされるカーネル関数の例を提示する．ただし，提示する修正案はユーザが記述したカーネル関数の意図を意識していないため，ユーザの意図とは全く異なる修正案を提示する可能性がある．本研究が支援する範囲では，限界まで最適化を行いたいというユーザの要求を満たすことはできない．ユーザは提示された修正案を利用する場合，プログラムの意味を考えアクセスパターンに合わせたデータの並び替えをしなければならない．

カーネル関数の修正例を示す．図 3 に修正前のカーネル関数の例を示し，図 4 に修正後のカーネル関数の例を示す． 図 3 のプログラムでは，5 行目の `d_b` に対する読み込みを行う先頭スレッドが 64 バイト境界にアライメントされていないためコアレッシ

```
1  __global__ void
2  kernel2(int *d_a, int *d_b)
3  {
4    int id = threadIdx.x + blockDim.x * blockIdx.x;
5    d_a[id] *= d_b[id + 16];
6  }
```

図 4: 修正後のカーネル関数

```
1  const int N = 512 * 512;
2  int size = sizeof(int) * N;
3  //デバイスメモリにメモリ領域を確保する
4  cudaMalloc((void**)&d_A, size);
5  cudaMalloc((void**)&d_B, size);
6  //配列の添字を記憶する変数の領域を確保
7  cudaMalloc((void**)&dst1, size); //書き込み先1
8  cudaMalloc((void**)&src1, size); //読み込み先1
9  cudaMalloc((void**)&src2, size); //読み込み先2
10 //添字記憶用変数の初期化
11 cudaMemset(dst1,-1,size);
12 cudaMemset(src1,-1,size);
13 cudaMemset(src2,-1,size);
14 //デバイスメモリにデータを書き出す
15 cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice);
16 //グリッドとブロックの次元を設定する
17 dim3 grid(512,1), block(512,1,1);
18 //カーネル関数の呼び出し（添字記憶用変数を引数に加える）
19 kernel_func<<<grid, block>>>
20           (d_A, d_B, dst1, src1, src2);
21 //ホストメモリにデータを書き戻す
22 cudaMemcpy(h_B,d_B,size,cudaMemcpyDeviceToHost);
23 //記憶した添字をホストメモリへ書き戻す
24 cudaMemcpy(h_dst1,dst1,
25           size,cudaMemcpyDeviceToHost);
26 cudaMemcpy(h_src1,src1,
27           size,cudaMemcpyDeviceToHost);
28 cudaMemcpy(h_src2,src2,
29           size,cudaMemcpyDeviceToHost);
```

図 5: プロファイリング用の命令を追加したプログラム

ングされない．図 4 に示す修正案は，アクセス先が 64 バイト境界にアライメントされるように d_b の添字を id+16 に変更する．

4. 実行例

提案手法を用いたプロファイリングと性能低下要因の探索を行い，ユーザにその原因を示す図とカーネル関数の修正案を提示するまでの過程を手動で実行した結果を示す．使用したグラフィックボードは GeForce 9800 GT であり，コンピュートケイパビリティは 1.1 である．

図 5 は，プロファイリングするために命令を追加した後のプログラムであり，図 6 は，図 5 の 19,20 行目で呼ばれたカーネル関数の内容である．図 5 のプログラムは，添字記憶用変数の領域を確保する命令と添字記憶用変数をホストにコピーする命令を追加し，カーネル関数の引数に添字記憶用変数を加えた．if 文などによってグローバルメモリへのアクセスが行われない場合を判別するために添字記憶用変数は-1 で初期化しておく．図 6 のプログラムは，グローバルメモリにアクセスするさいの各配列の添字を記憶する命令を追加した．

プログラムを実行すると，プロファイリング結果が分析され，グローバルメモリへのアクセスがコアレッシングされたか，されなかったかが出力される．図 5 のプログラムを実行した後の出力の一部を図 7 に示す．図 7 でアドレスが unknown と表示している理由は，カーネル関数内の if 文によって添字を記憶する命令が実行されなかったからである．

図 7 の A，C では，転送するデータ型サイズとコアレッシングされなかった部分のスレッド番号，アクセス先アドレス，コア

```
1  __global__ void
2  kernel_func(int *d_A,int *d_B,
3              int *dst1,int *src1,int *src2)
4  {
5    int id = threadIdx.x + blockDim.x * blockIdx.x;
6    if(id % 2 == 0){
7      d_B[id] = d_A[id] + d_A[id + 1];
8      //各アクセス先の添字を記憶する
9      dst1[id] = id;
10     src1[id] = id;
11     src2[id] = id + 1;
12   }
13 }
```

図 6: プロファイリング用の命令を追加したカーネル関数

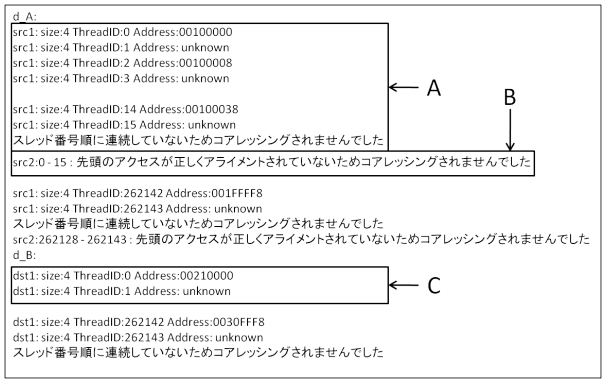

図 7: 出力

レッシングされなかったことを示すメッセージを出力している．図 8 は出力 A から得られた変数 d_A に対する 1 つ目の読み込みのメモリアクセス図であり，図 9 は出力 C から得られた変数 d_B に対する書き込みのメモリアクセス図である．それぞれのメモリアクセス図より，変数 d_A に対する 1 つ目の読み込と変数 d_B に対する書き込みは，スレッド番号が偶数のスレッドのみグローバルメモリへのアクセスを行っていることがわかる．よって，変数 d_A からの 1 つ目の読み込み，および変数 d_B への書き込みは，グローバルメモリへのアクセスがスレッド番号順に連続していないためコアレッシング条件の 2 つ目を満たしていない．

図 7 の B では，コアレッシングされなかったスレッド番号の範囲とコアレッシングされなかったことを示すメッセージを出力している．図 10 は出力 B から得られた変数 d_A に対する 2 つ目の読み込みのメモリアクセス図である．図 10 より，スレッド番号が偶数のスレッドのみグローバルメモリへアクセスを行っていることに加え，先頭スレッドがアクセスするアドレスが 0x00100004 であるため，64 バイト境界にアライメントされていないことがわかる．よって，変数 d_A からの 2 つ目の読み込みは，グローバルメモリへのアクセスがスレッド番号順に連続しておらず，先

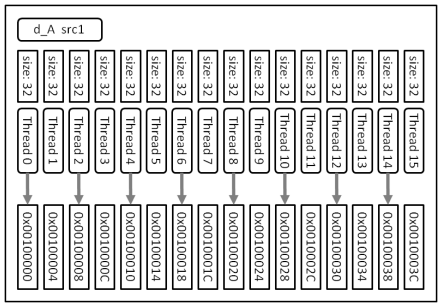

図 8: 1 つ目の読み込みのメモリアクセス図

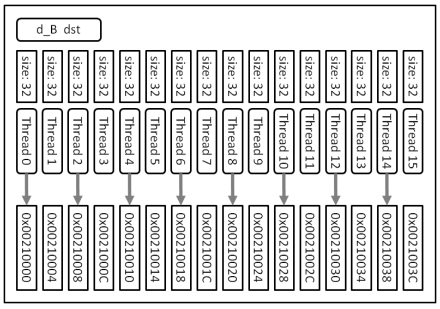

図 9: 書き込みのメモリアクセス図

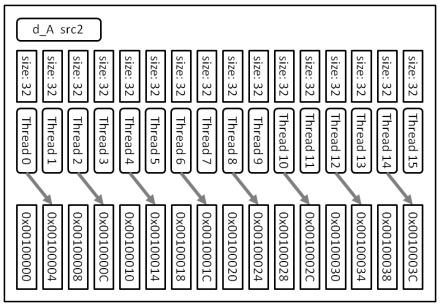

図 10: 2 つ目の読み込みのメモリアクセス図

頭スレッドのアクセス先アドレスが 64 バイト境界にアライメントされていないため，コアレッシング条件の 2 つ目と 3 つ目を満たしていない．

上記からわかるように，図 6 のプログラムで問題となっていることは，偶数番目のスレッドだけがメモリアクセスを行うためにメモリアクセスが連続しないことと，2 つ目の読み込み先が 64 バイト境界，または 128 バイト境界にアライメントされていないことの 2 つである．2 つの問題を解決するためには，偶数番目のスレッドだけがメモリアクセスを行うのではなく，一定の数のスレッドが連続してメモリアクセスを行うようにアクセス幅を調整しなければならない．一定の数は，コアレッシングされるグローバルメモリへのアクセスに必要なスレッド数が 16 本であることから，16 の倍数であることが望ましい．アクセス幅が 16 の場合，転送するデータ型サイズが 4 バイトであれば 64 バイト境界にアライメントされ，8 バイトであれば 128 バイト境界にアライメントされるため，2 つの問題を一度に解決できる．

以上から，アクセス幅を 16 の倍数に修正することでコアレッシングされるグローバルメモリへのアクセスが行われると期待できる．

図 11 に，図 6 のカーネル関数の修正案を示す．図 11 は，アクセス幅を 32 に修正している．図 6 の 6 行目の分岐条件を，2 で割った余りが 0 のときではなく，64 で割った余りが 32 未満のときに処理を行うように変更している．32 スレッドずつメモリアクセスを行うため，コアレッシングされるグローバルメモリへのアクセスが行われる．この修正案を利用する場合，使用するデータをアクセスパターンに合わせて適切な位置と入れ替える命令を記述する必要がある．

```
1  __global__ void
2  kernel_func(int *d_A, int *d_B)
3  {
4    int id = threadIdx.x + blockDim.x * blockIdx.x;
5    if(id % 64 < 32)
6      d_B[id] = d_A[id] + d_A[id + 32];
7  }
```

図 11: カーネル関数の修正案

5. おわりに

本稿では，CUDA プログラムをプロファイリングして性能低下要因を探索し，プログラムの開発と最適化を支援する手法を提案した．提案手法では，コアレッシングされないグローバルメモリへのアクセスをプロファイリングし，転送するデータ型サイズとグローバルメモリへのアクセス先アドレスを取得してコアレッシング条件と照らし合わせることで性能を低下させる原因を探索する．探索した結果を出力し，コアレッシングされないグローバルメモリへのアクセスが見つかった場合は，満たされなかったコアレッシング条件を示すためにスレッドとアドレスの関係図を表示する．ユーザが記述したアクセスパターンと類似する，コアレッシングされるグローバルメモリへのアクセスを行うカーネル関数の例を提示する．

関連研究として，CUDA プログラムを動的に解析し，バンクコンフリクトを検知して CUDA プログラムの開発を支援することを目的とした Boyer らの研究 [5] がある．この研究は，共有メモリへのアクセスが性能を低下させる原因となっている場合の開発支援であり，本研究はグローバルメモリへのアクセスが原因となっている場合の開発支援である．グローバルメモリへのアクセス時間は共有メモリへのアクセス時間よりも長い．よって，本研究は性能低下要因を解消した効果で優れていると言える．

提案手法では，探索可能な性能を低下させる原因はカーネル関数内におけるコアレッシングされないグローバルメモリへのアクセスに限られているため，他の原因の探索を可能にすることが今後の課題として挙げられる．本稿で述べたプロファイリング用の命令の追加，コアレッシング条件の照合，およびメモリアクセス図の作成，カーネル関数の修正案提示はすべて手動で行っているため，性能を低下させる原因を自動で探索するツールを作成することも今後の課題である．コンピュートケイパビリティ 1.2 以降では，すべてのグローバルメモリへのアクセスがコアレッシングされる．しかし，グローバルメモリへアクセスするさいのアドレスのアライメントなどを考慮しなければ GPU の性能を引き出すプログラムは記述できないため，コアレッシングに関する最適化支援は今後も必要である．

参考文献

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[2] General-purpose computation on graphics hardware. http://gpgpu.org/.

[3] NVIDIA Corporation. *CUDA C Programming Guide Version 3.1.1*, July 2010.

[4] S. Ryoo. Program optimization strategies for data-parallel many-core processors. *UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN*, page 3314878, 2008.

[5] M. Boyer, K. Skadron, and W. Weimer. Automated dynamic analysis of CUDA programs. In *Workshop on Software Tools for MultiCore Systems*, 2008.