Vol.2012-HPC-134 No.5
2012/6/1

GPUにおけるダイバージェンス削減による高速化手法

加藤 誠也[1,a)]　須田 礼仁[1,b)]　玉田 嘉紀[1,c)]

概要：近年 GPU は計算能力において目覚しい発展を続けており，NVIDIA の CUDA C に代表される演算用の言語の導入などによって，科学技術計算分野において重要な役目を担うようになっている. その一方で，CUDA のプログラミングモデルである SIMT(Single Instruction Multiple Threads) の特徴として，ダイバージェンスと呼ばれる問題があり，GPU の実行率が低下するため，GPU は CPU に比べると条件分岐の影響を受けやすい. そのため，条件分岐の最適化がより重要になっている. 本論文では，GPU のダイバージェンスを削減し，実行率を向上させるための手法として，動的割り付け・分岐統一化の 2 つの手法を提案する. 動的割り付けは主にデータごとに長さの異なるループが実行されるカーネルに対して適用可能である. これは，CUDA におけるブロック単位でデータを割り当て，ブロック内で各スレッドに動的にデータを割り当てることで，各スレッドに割り当てられるデータの処理量を均等にし，GPU の実行率を高める手法である. 分岐統一化はデータに応じた条件分岐によって処理の大半が分かれているカーネルに対して適用可能である. これは，各スレッドに複数のデータを割り当てて，ある分岐方向の処理を行う際に，各スレッドが自分の持つデータの中からその方向に分岐するデータを選んでそれに対して処理を行うようにすることで，各スレッドに各条件分岐の実行中に実行するデータがあるようにして，GPU の実行率を上げることができるという手法である. これらの手法の有効性は，サンプルコードを用いた実験によって確認した.

キーワード：GPU, CUDA, ダイバージェンス, 最適化

Optimization Techniques for Reducing Branch Divergence on GPUs

Seiya Kato[1,a)]　Reiji Suda[1,b)]　Yoshinori Tamada[1,c)]

Abstract: Recently, GPUs have progressed tremendously in computational power. Thus, the role of GPUs has become important in the field of computational science with the introduction of programing languages for GPU computation such as NVIDIA CUDA C. On the other hand, a problem called branch divergence has appeared as the feature of the programming model of CUDA called SIMT (Single Instruction Multiple Threads). Because of this, GPUs are more likely to be affected by conditional branch instructions than CPUs. Therefore, optimization of conditional branch is very important on GPUs in order to utilize the entire computational power. This paper proposes two techniques for reducing branch divergence on GPUs. *Dynamic work assignment* is applicable when almost every part of the kernel is a loop whose number of iterations is different with respect to input data. This technique increases the GPU execution rate by assigning data to each CUDA block and assigning data to each thread dynamically in the block so that the amount of computation of each thread becomes equal to others in the block. *Branch path unification* is applicable when almost every part of the kernel executes the different branch path by a conditional branch depending on data. This technique increases the GPU execution rate by assigning multiple data to a thread and exchanging the order of data assigned to threads so that the same branch path is executed by as many threads as possible and all the branch paths are executed one after the other. The effectiveness of these techniques has been confirmed by the experiments with the sample codes.

Keywords: GPU, CUDA, Divergence, Optimization

1. はじめに

近年 GPU(Graphic Processing Unit) はその計算能力において大きな発展を遂げて来た．また，NVIDIA 社の TESLA に代表される様な GPGPU(General-Perpose computing on Graphic Processing Unit) 用の GPU の登場や，CUDA C の様な GPU 向け汎用計算用の言語の導入によって，GPU を汎用計算に用いるための環境が整えられ，様々な目的に利用されるようになってきた．

その一方で, GPU の計算能力を余すことなく発揮させることは難しく, 様々な問題点がある．その 1 つとして, CUDA のプログラミングモデルである SIMT(Single Instruction Multiple Threads) における”Branch Divergence”(以下, ダイバージェンスと示す) と呼ばれる問題がある [1]．CUDA ではスレッドは一定個数毎に warp という単位に割り当てられており，1 つの warp 内のスレッドはすべて同じ命令が発行されるようになっている．そのため，条件分岐文によって warp 内のスレッドが異なった命令を実行することになった場合，他のスレッドを休止してその命令を実行するようになっているため，その分 GPU の使用率が低下してしまう．これが，ダイバージェンスという問題である．本稿では，特定の条件下においてこの影響を低減することが可能な手法として，2 つの手法を提案する．尚，本稿のプログラムコードは CUDA C を想定している．

1 つ目は動的割り付けと呼んでいるもので, スレッド毎のループ処理の繰り返し回数の差異によって生じるダイバージェンスを減らすための手法である．この手法は，カーネルにおいて，ほとんどの処理が繰り返し回数が入力データに応じて変わるようなループ処理によって占められているような場合において有効である．データを各スレッドに静的に与えるのではなく，ブロック又は warp 単位にそのスレッド数よりも多くのデータを割り当てて，実行時に処理が終わったものから動的に次のデータを取得できるようにすることが出来る様にする．こうすることによって，スレッド間の処理量をなるべく均等にし，一部のスレッドに処理量が偏ることによって発生するダイバージェンスを減らす．

2 つ目は分岐統一化と呼んでいるもので, if-else 命令などによってスレッド毎に実行するプログラムの内容がばらけることによって生じるダイバージェンスを減らすための手法である．この手法は，カーネルにおいて，ほとんどの処理が 1 つの if-else 文や switch 文などの分岐文中で行われているような場合において有効である．データを各スレッドに 1 つ与えるのではなく，各スレッドに複数のデータを与え，その中から処理するデータを任意の順で取り出せる様にすることで，ある分岐方向の処理を実行する際に各スレッドに処理するデータがなるべく存在するようにしている．こうすることによって，あるスレッドが通らない分岐方向の処理中にはそのスレッドは休止状態になるという仕様によって発生するダイバージェンスを減らしている．

これらの手法の有効性を示すために，簡単な検証用のコードを用意し，実験を行ったところ，これらの手法を利用した方が計算が速くなる場合があることが分かった．従って，これらがダイバージェンスの影響を減らすのに有効な手法であることが確認できた．

2. 関連研究

ダイバージェンスの発生する命令数を減らそうとしているものに Tianyi David Han らによる Branch Distribution がある [2]．これは，マルチパス分岐文中の共通の命令を分岐していない所に括り出すことによって，ダイバージェンス中の命令数を減らし，実行効率をあげるという手法である．同じ様な系統のものとして，Imen Chakroun らによる Branch Refactoring がある [3]．これは，sign などの算術命令を利用することで，条件分岐文を減らすという手法である．また，Eddy Z. Zhang ら [4] の研究にホストコンピュータの CPU を利用してデータを整列し，ダイバージェンスの発生を減らすというものがある．

ダイバージェンスをプログラマに発見しやすくするための手法も提唱されている．Bruno Coutinho らによる”Divergence map”[5] はダイバージェンスの発生箇所と量を可視化し，プログラマにどこでパフォーマンスが低下しているかをわかりやすくするツールである．

ダイバージェンスによる機能低下を減らすための新たなハードウェアの追加の提案もある．Dynamic Warp Subdivision[6] は warp 内でダイバージェンスが発生した際に warp を動的に分割するという手法である．Dynamic Warp Formation[7], [8] は，ダイバージェンスが発生する際に warp を動的に再編成し，ダイバージェンスを起こさないようにするといったものである．

本稿の手法は，複数のデータを一つのスレッドに割り当てるというものだが，同じ様に複数のデータを一つのスレッドに割り付ける手法として Vasly Volkov[9] のものがある．これは，そうすることで実行されるスレッドの数を減らして，各スレッドで重複することになる内容のレジスタ使用数を減らしつつ，スレッド内で実行に十分な並列性を確保することを目的としている．Snaider Carrillo らによる”Branch Splitting”[10] は，if-else 条件分岐文がカーネルの大部分を占めている際に適用可能な手法である．これは，そのような場合に if 側と else 側にカーネルを分割して両方を実行することによって，レジスターの使用数を減

[1] 東京大学大学院 情報理工学系研究科 コンピュータ科学専攻
Graduate School of Information Science and Technology The University of Tokyo
[a)] katose@is.s.u-tokyo.ac.jp
[b)] reiji@is.s.u-tokyo.ac.jp
[c)] tamada@is.s.u-tokyo.ac.jp

```
__global__ kernel(){
   preparation();
   for(int i = 0;i < スレッド毎に異なる数;i++;)
      function();
   settlement();
}
```

図 1 動的割り付けに適したカーネルコード

Fig. 1 An example of kernel code suitable to the dynamic work assignment method.

```
i = data[blockIdx.x * blockDim.x + threadIdx.x];
preparation();
while(branch(i)){
   function();
   i = next(i);
}
settlement();
```

図 4 動的割り付け適用前の擬似カーネルコード.

Fig. 4 Pseudo kernel code before using dynamic work assignment method.

らすことによって実行性能を向上させることを目的としている.

3. 提案手法

3.1 動的割り付け

「動的割り付け」はスレッド間の繰り返し回数の差異から生じるダイバージェンスを減らすことを目的とした手法である．この手法は**図 1** の様に，GPU カーネルの大部分がループ処理によって占められており，その処理の回数がスレッド毎に異なっているような場合において有効である．

ある warp に 6 つのスレッドがある場合における，前述のプログラムの実行を擬似的に表したものが**図 2** である．縦軸は同じ warp 内でのスレッドを表し，横軸は時間経過で各スレッドで処理されていく内容を示している．GPU コアから見た場合は処理を行っている warp でメモリアクセスなどによる待ち時間が生じた際には，他の warp に処理を切り替えるが，この図では warp から見た場合の処理される様子を示しているため，切り替えについては無視している．また，本手法を用いる際に複数のデータを 1 つの warp に割り当てる必要があるため，3 つの warp の処理の様子を横軸に並べてある．これらの warp は数が少ない場合は一度に並列に処理される可能性があるが，今回は GPU がすべて並列に処理しきれないだけの warp 数になるだけのスレッドが一度に実行されているとしている．図中の緑の領域はループ処理の前の処理時間 (図 1 では prepare にあたる)，赤の領域は 1 回のループ処理時間 (図 1 では function にあたる)，青の領域はループ後の処理時間 (図 1 では settlement にあたる)，白い領域は何もしていない時間を表している．この図の様に，warp 内でループ処理の回数が少ないスレッドは最もループ処理の回数の多いスレッドの計算が終了するまで次の処理を行うのを待つ必要があるため，何もしていない時間が発生し，実行効率が下がってしまっている．

本手法では，この図に置ける空欄の量を減らすため，各ブロック或いは warp 毎にその中のスレッドの数よりも多くのデータを割り当てて，ループ処理の終わったスレッドから次のデータを動的に取得可能なようにする．こうすることによって，各スレッドの処理量がなるべく等しくなるようにしている．**図 3** は図 2 の処理にこの手法を用いて warp 毎に 18 個のデータを割り当てた場合の処理の様子を擬似的に表したものであり，新に発生した黄色の領域はその動的なデータの割り当てに必要な時間を表している．両図の赤い領域内に示されたアルファベットはそれぞれ対応しており，本手法の適応によって各スレッドで処理されるデータが変化していることを示している．図 3 では図 2 に比べて空白領域が減り，実行効率が改善していることが見て取れる．その分新たな処理が発生してはいるが，この実行効率の改善によってその処理を上回る高速化が可能であるような場合には，これらの図の様に処理の高速化が期待できる．

図 4 は入力データに応じた異なった回数のループ処理と，共通のその前後の処理からなるカーネルの擬似コードであり，**図 5** はそれに各ブロック毎に”DATA_PER_BLOCK”個のデータを割り当てる形で本手法を適用した後の擬似コードを表している．そのため，同じ数のデータを処理するには，適用後のコードを実行する際に適用前のコードの”DATA_PER_BLOCK”分の 1 のスレッド数で実行する必要がある．これらの擬似コードでは，各ブロック及びスレッドは 1 次元領域に配置されているとしており，入力データは data 変数に同様に一次元に配置されているとしている． ループの前の処理を”preparation”，ループ内で繰り返される関数を”function”，ループ後の処理を”settlement”，ループ回数を決める関数を”branch”，条件を更新する関数を”next”としている．このコードでは”counter”という変数を Shared Memory 上に確保し，そこにどのデータまで処理が行われたかを記録することによって各スレッドが処理するデータが重複しないよう管理している．次に処理するデータの番号を取得する際には，この”counter”に各スレッドが同時にアクセスしないように読み込み，インクリメント，書き戻しを行う必要があるため，”atomicAdd”機能を利用している．warp 毎にデータを割り当てるようにする際には，”counter”をブロック毎の warp 数分だけ用意すればよい.

3.2 分岐統一化

「分岐統一化」は条件分岐文によるスレッド毎の実行内容の違いから生じるダイバージェンスを，減らすことを目的とした手法である．この手法は**図 6** の様に，GPU カー

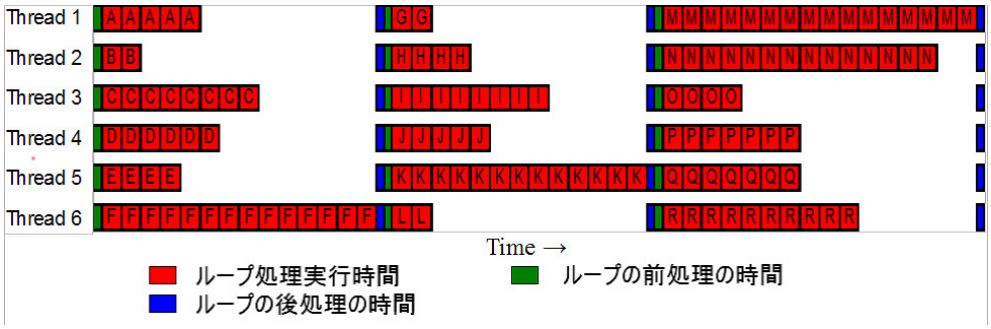

図 2 動的割り付け適用前のカーネルの擬似的な実行の様子.
アルファベットは図 3 とのデータの対応を示している.

Fig. 2 The pseudo flow of the execution of a kernel before applying the dynamic work assignment method. Alphabets corrspond to those of Figure 3.

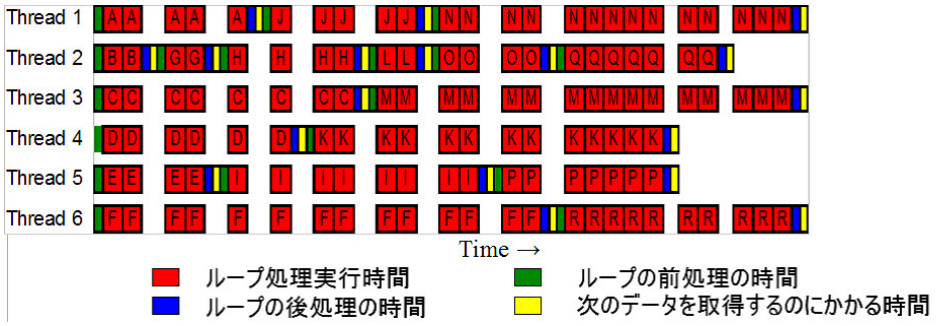

図 3 動的割り付け適用後のカーネルの擬似的な処理の様子.
アルファベットは図 2 とのデータの対応を示している.

Fig. 3 The pseudo flow of the execution of a kernel after applying dynamic work assignment method. Alphabets corrspond to those of Figure 2.

```
__shared__ counter;
if(threadIdx.x == 0)
   counter = blockDim.x;
i = data[blockIdx.x * blockDim.x + threadIdx.x];
__syncthreads();
preparation();
while(1){
   (while(!(branch(i)))){
      settlement();
      //次のデータ番号を取得し，カウンタをインクリメント.
      int next_data = atomicAdd(&counter,1);
      if(next_data ≥ DATA_PER_BLOCK)
         return;
      i = data[blockIdx.x * blockDim.x + next_data];
      preparation();
   }
   function();
   i = next(i);
}
```

図 5 動的割り付け適用後の擬似カーネルコード.

Fig. 5 Pseudo kernel code after applying the dynamic work assignment method.

ネルの大部分が if-else 文や switch 文などの条件分岐によって分けられており，その分岐方向がスレッド毎に異なっているような場合において有効である.

ある warp に 6 つのスレッドがあるとした GPU におけ

```
__global__ kernel(){
   preparation();
   if(スレッド毎に異なる条件)
      functionA();
   else
      functionB();
   settlement();
}
```

図 6 分岐統一化に適したカーネルコード.

Fig. 6 The kernel code suitable to the branch unification method.

る各スレッドにおける前述のプログラムの実効の様子を擬似的に表したものが**図 7** である．縦軸は同じ warp 内でのスレッドを表し，横軸は時間経過で各スレッドで処理されていく内容を表している．前章同様に warp の切り替えについては無視している．また，本手法を用いる際に複数のデータを 1 つの warp に割り当てる必要があるため，8 つの warp の処理の様子を横軸に並べてある．これらの warp は数が少ない場合は一度に並列に処理される可能性があるが，今回は GPU がすべて並列に処理しきれないだけの warp 数になるだけのスレッドが一度に実行されているとしている．図中の緑の領域は分岐処理前の処理時間 (図 6 では prepare にあたる)，赤の領域は条件を満たす方に分岐

```
i = data[blockIdx.x * blockDim.x + threadIdx.x];
preparation();
if(branch(i)){
  functionA();
else
  functionB();
}
settlement();
```

図 9 分岐統一化適用前の擬似カーネルコード.

Fig. 9 Pseudo kernel code before applying the branch unification method.

した際の処理時間 (図 6 では functionA にあたる)，灰色の領域は条件を満たさない方に分岐した際の処理時間 (図 6 では functionB にあたる)，青の領域は分岐処理の後の処理時間 (図 6 では settlement にあたる)，白い領域は何もしていない時間を表している．複数のパスを持つ条件分岐文では，すべての warp 内のスレッドが合流するまで分岐した先の命令がそれぞれ順に実行されるが，各 warp 内では同時にひとつの命令しか発行できないため，ある分岐先が実行されている間，そちらに分岐していないスレッドはこの図の様に待ち状態となってしまう．そのため，例えばこの様に均等に 2 つのパスに分岐する様なカーネルでは分岐中の実行効率は半分になってしまう．

本手法では，これらの空白領域を減らすために，[9] の様に，1 つのスレッドに複数のデータを割り当て，あるパスを実行する際に，各スレッドがその中からそのパスに分岐するものを選び，それに対して処理を実行する様にする．こうすることによって，あるパスの実行時に各スレッドに処理すべきデータがなるべく存在するようにしている．**図 8** は図 7 の処理にこの手法を用いてスレッド毎に 8 個のデータを割り当てた場合の処理の様子を擬似的に表したものであり，新に発生した黄色の領域は処理するデータの選択に必要な時間を表している．両図の各スレッドにおける赤，灰色の領域の数字はそれぞれ対応しており，各スレッドで同時に処理されるデータが変化していることを示している．図 8 では図 7 に比べて空白領域が減り，実行効率が改善していることが見て取れる．その分新たな処理が発生してはいるが，この実行効率の改善によってその処理を上回る高速化が可能であるような場合には，これらの図の様に処理の高速化が期待できる．

図 9 は 1 つの if-else 条件文と共通のその前後の処理からなる単純な擬似コードであり，このコードに各スレッドに”DATA_PER_THREAD”個のデータを割り当てて，本手法を適用した後の擬似コードが**図 10** である．適用後のコードでは 1 スレッドで”DATA_PER_THREAD”個のデータを処理しているため，同数のデータを処理するには，適用後のコードを実行する際には適用前のコードの”DATA_PER_THREAD”分の 1 のスレッド数で実行する必要がある．これらの擬似コードでは，各ブロック及び

```
counterA = 0;
counterB = 0;
data = &(data[(blockIdx.x * blockDim.x + threadIdx.x)
   * DATA_PER_THREAD]);
while(1){
  //if 文側
  i;
  data_exist = false;
  while(counterA < DATA_PER_THREAD){
    i = data[counterA];
    counterA++;
    if(branch(i)){
      data_exist = true;
      break;
    }
  }
  if(data_exist == true){
    preparation();
    functionA();
    settlement();
  }
  //else 文側
  data_exist = false;
  while(counterB < DATA_PER_THREAD){
    i = data[counterB];
    counterB++;
    if(!(branch(i))){
      data_exist = true;
      break;
    }
  }
  if(data_exist == true){
    preparation();
    functionB();
    settlement();
  }
  //すべてのデータの処理が終わっていたらループを終了する.
  if(counterA ≥ DATA_PER_THREAD
    && counterB ≥ DATA_PER_THREAD)
    return;
}
```

図 10 分岐統一化適用後の擬似カーネルコード.

Fig. 10 Pseudo kernel code after applying the branch unification method.

スレッドは 1 次元領域に配置されているとしており，入力データは data 変数に同様に一次元に配置されているとしている．条件分岐の前の処理を”preparation”，条件成立時の処理を”functionA”，条件非成立時の処理を”functionB”，条件分岐終了後の処理を”settlement”，条件成立判定を行う関数を”branch”としている．このコードでは，各スレッドに”counterA”，”counterB”という 2 つの変数を用意し，これらに何番目までのデータを処理したのかを if 文側，else 文側それぞれ保存することによって，処理を管理している．あるパスを実行する際には，対応した変数を用いて，未処理かつ該当方向へ分岐するデータを探し，適合するものが存在する場合にはそれに対して処理を行っている．また，このコードでは if 側への分岐と else 側への分岐を交互に実

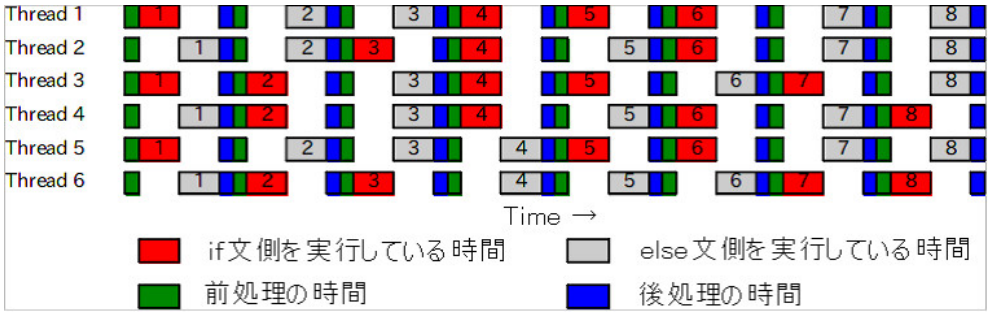

図 7 分岐統一化適用前のカーネルの擬似的な処理の様子．
数字は図 8 とのデータの対応を示している．

Fig. 7 The pseudo flow of the execution with a kernel before applying the branch unification method. Numbers corrspond to those of Figure 8.

図 8 分岐統一化適用後のカーネルの擬似的な処理の様子．
数字は図 7 とのデータの対応を示している．

Fig. 8 The pseudo flow of execution with a kernel after applying the branch unification method. Numbers corrspond to those of Figure 7.

行しているが，それぞれへの分岐確率が異なっていることが分かっている場合には，それに応じて実行頻度を変えることも考えられる．

4. 実験

4.1 実行環境

実行環境は以下の通りである．
CPU:Intel(R) Core(TM)i7 920 2.67Ghz
(インテルターボブーストテクノロジーは無効化)
メモリ:DDR3 3GB
GPU:NVIDIA GEFORCE GTX 560Ti
(GPU メモリ:GDDR5 1024MB)
OS:Windoes 7 Home Premium 64bit
コンパイラ:Microsoft Visual Studio C++ 2010 Express Edition
CUDA:CUDA Toolkit 4.1

4.2 動的割り付け

実験のために**図 A·1** の様なプログラムを用意した．処理の主となるループの回数は入力データに応じてスレッド毎に違う回数実行されるようになっており，本実験で

```
function(int tmp){
#pragma unroll 128
  for(int i = 0;i < LOOP_NUM;i++){
    tmp = 0xffff & (tmp * tmp + tmp);
  {
  return tmp;
}
```

図 11 動的割り付けの実験に用いた関数．

Fig. 11 The code used in the experiment for the dynamic work assignment method.

はその回数は 2048 回から 8192 回の間で乱数を用いて決定した．kernel は本手法適用前のコードであり，各スレッドは 1 つのデータを処理し，ループに関しては warp 内でもっとも長い物が終了するまで待つようになっている．kernelEnhanced は本手法適用後のコードであり，各ブロックに”ARRAY_PER_BLOCK”(本実験では 1024 とした) のデータが割り当てられており，ブロック内のスレッドで動的に割り付けられるようになっている．図 1 における”function”にあたる関数は**図 11** の様になっており，”LOOP_NUM”の値に応じて処理量を変更出来る様になっている．本実験ではこの値を変更しながら適用前と後でのカーネルの実行に必要な時間を測定した．適用後のも

表 1 動的割り付け実験における手法適用前後の”LOOP_NUM”と処理時間

Table 1 The comparison of the consumed times of the experiment with the dynamic work assignment method with respect to LOOP_NUM.

LOOP_NUM	適用前	適用後
1	0.003578s	0.006199s
2	0.004829s	0.006539s
4	0.005982s	0.007417s
8	0.010972s	0.011640s
16	0.021037s	0.020873s
32	0.044515s	0.044844s
64	0.085638s	0.081224s
128	0.172095s	0.153413s
256	0.327804s	0.301698s
512	0.655889s	0.593056s

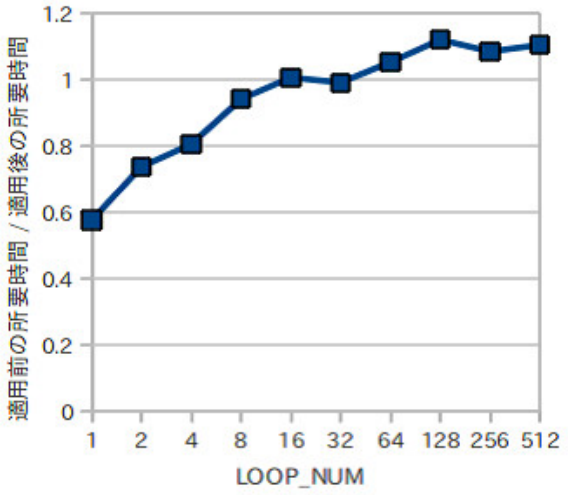

図 12 動的割り付け適用前後の所要時間比.

Fig. 12 The rate of consumed time between before and after the optimization with the dynamic work assignment method.

のでブロックの数は 32，各ブロック内のスレッドの数は 128 として実行し，適用前のものについては各ブロックで処理するデータ数が 1024/128=8 分の 1 のため，ブロックの数を 8 倍として実行した．適用後のコードでは，グローバルメモリへのアクセスが各スレッドでバラバラになるため，それ改善をするために，一度結果を Shared Memory 上に保存し，最後にグローバルメモリに保存するようにしている．

表 1 が実際に所要時間を測定した結果であり，**図 12** は横軸に LOOP_NUM，縦軸に所要時間の比を取ってそれをグラフ化したものである．データを選択する処理のある分ループ処理の負荷が小さいときには本手法適用後のものの方が実行が遅くなっているが，ループ処理の負荷がある程度以上ある時には適用後のものの方が所要時間が短くなっている．LOOP_NUM を 8192 まで大きくした際の所要時

```
function(int tmp){
#pragma unroll 100
  for(int i = 0;i < LOOP;i++){
    tmp = 0xffff & (tmp * tmp + tmp);
  {
  return tmp;
}
```

図 13 分岐統一化の実験に用いた関数.

Fig. 13 The code used in the experiment for the branch unification method.

間は適用前で 11.329095s，適用後で 9.348179s であり，約 21%の改善が見られた．

先ほどの様にループの回数の分布が均一なものではこの手法の効果は大きくは無いが，ループ回数が短いものに偏っている場合はこの手法によって大きな改善が見込める．先ほどの実験において，データ毎のループ回数を 90%は 1 から 2048 まで，残り 10%は 2048 から 8192 までと大きく偏らせた場合，LOOP_NUM が 128 の時，適用前のものの所要時間は 0.169481s，適用後のものは 0.065186s となり，約 160%改善され，先ほどの LOOP_NUM が 128 の時の 12%に比べて大きく改善された．

4.3 分岐統一化

実験のために**図 A·2** の様なプログラムを用意した．if-else 文での分岐方向は乱数で発生させた入力データの下から 3bit 目の有無に応じて分岐するようになっている．kernel は本手法適用前のコードであり，各スレッドは 1 つのデータを処理し，先ほどの条件にしたがって functionA，functionB のいずれかを実行している．kernelEnhanced は本手法適用後のコードであり，各スレッドに”DATA_PER_THREAD”個のデータが割り当てられており，各スレッドはそこから順次分岐条件を満たすもの，満たさないものを選択してそれに対して functionA，functionB を実行している．図 6 における”functionA”，”functionB”は 4.2 章と同様に**図 13** の様になっており，”LOOP”の値に応じて処理量を変更出来る様になっている．本実験ではこの値と DATA_PER_THREAD を変更しながら適用前と後でのカーネルの実行に必要な時間を測定した．適用後のものでブロックの数は 64，各ブロック内のスレッドの数は 128 として実行し，適用前のものについては各スレッドで処理するデータが DATA_PER_THREAD 分の 1 のため，ブロックの数を DATA_PER_THREAD 倍することでスレッド数を増やして実行した．

表 2 は本手法適用前のものについての所要時間の測定結果，**表 3** は本手法適用後のものについての所要時間の測定結果であり，**図 14** はそのうち DATA_PER_THREAD=64 の時について，横軸に LOOP 縦軸に所要時間の比を取ってグラフ化したものである．データを選択する処理のある分，分岐中の処理の負荷が小さい時には本手法適用後のものの方が実行が遅くなっているが，ループ処理の負荷がある程度

表 2　分岐統一化適用前の所要時間.
Table 2 The consumed time before the optimization with the branch unification method.

LOOP ＼ DATA_PER_THREAD	1	4	16	64
1	0.000041s	0.000052s	0.000174s	0.001356s
10	0.000043s	0.000055s	0.000172s	0.001353s
100	0.000058s	0.000087s	0.000240s	0.001449s
1000	0.000144s	0.000404s	0.001364s	0.004465s
10000	0.001015s	0.004047s	0.013259s	0.043370s

表 3　分岐統一化適用後の所要時間.
Table 3 The consumed time after the optimization with the branch unification method.

LOOP ＼ DATA_PER_THREAD	1	4	16	64
1	0.000038s	0.000040s	0.000043s	0.000059s
10	0.000035s	0.000042s	0.000049s	0.000078s
100	0.000041s	0.000054s	0.000105s	0.000303s
1000	0.000142s	0.000384s	0.001338s	0.005165s
10000	0.001119s	0.003426s	0.012963s	0.051003s

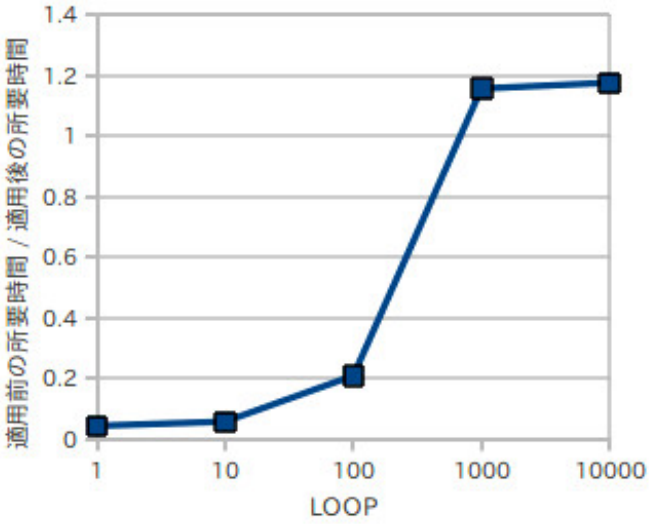

図 14　分岐統一化適用前後の所要時間比.
Fig. 14 The rate of consumed time of before and after the optimization with the branch unification method.

表 4　分岐統一化適用前後の分岐部分の実行率.
Table 4 The execution rate of the branch path before and after the optimization with the branch unification method.

適用前	50.0%			
適用後				
DATA_PER_THREAD	1	4	16	64
	50.0%	51.8%	66.6%	79.5%

以上ある時には適用後のものの方が所要時間がわずかではあるが，短くなっている．**表 4** は分岐部分におけるスレッドの稼働率を測定したものである．本手法適用前のコードでは各スレッドは if-else 文のうち if 文側か else 文側のどちらか片方しか実行しないため，実行率は 50%となっている．warp 内のすべてのスレッドが同方向にした場合が存在すれば，実行率は 50%よりも大きくなるが，そのような warp の出現確率は 2^{31} であり，乱数の制度にもよるが，まず存在しない．適用後のコードでは DATA_PER_THREAD が大きくなるほど改善している．しかし，処理すべきデータ数が同じである場合，DATA_PER_THREAD を大きくするほど並列性が低下してしまうため，本手法の使用時にはデータ数に応じた調整が必要となる．また，今回の様に分岐先が 2 つの場合は最大でも実行率は 50%の改善に留まるが，分岐先が多数に分かれている場合にはより効果が期待できる．

5. おわりに

本稿では GPU におけるダイバージェンス問題を低減するための手法として 2 つの手法を提案した．第 4 章においてそれらの有効性について簡単な検証用のプログラムを用意し，実験を行った結果，それらの手法が有効な場合が存在するということが分かった．しかしながら，これらの手法は適用可能な形のプログラムにおいて常に有効ということはなく，手法の実装によって発生する新たなコストをそれによる高速化が上回る場合を見極める必要がある．また，いずれの手法も 1 スレッドあたりの割り当てを増やすという方法を取っているため，それを行うにあたっては十分に並列に計算できるデータがあることが前提となってしまっている．

本稿の手法が適用可能な状況は非常に限られた場合のみであり，ダイバージェンスを減らすための新たな手法の探索は続けられる必要があると考えられる．また，手法の適用時に発生する様々な制約や条件も踏まえて，それを活かすためのライブラリやツールなどの開発も必要であると思われる．

謝辞 本研究の一部は JST CREST「進化的アプローチによる超並列複合システム向け開発環境の創出」の援助を受けています.

参考文献

[1] NVIDIA CUDA C Programming Guide Version 4.1, 2011.
[2] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. GPGPU-4, 1-8, 2011.
[3] Imen Chakroun, Ahcène Bendjoudi and Nouredine Melab. Reducing thread divergence in GPU-based B&B applied to the flow-shop problem. PPAM 2011, 1-10, 2011.
[4] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo and Xipeng Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. ICS '10, 115-126, 2010.
[5] Bruno Coutinho, Diogo Sampaio, Fernando M. Q. Pereira and Wagner Meira Jr. Performance debugging of GPGPU applications with the divergence map. SBAC-PAD '10, 33-40, 2010.
[6] Jiayuan Meng, David Tarjan and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. ISCA '10, 235-246, 2010.
[7] Wilson W. L. Fung, Ivan Sham, George Yuan and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. MICRO 40, 407-418, 2007.
[8] Wilson W. L. Fung, Ivan Sham, George Yuan and Tor M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. ACM Transactions on Architecture and Code Optimization, Volume 6 Issue 2, 1-37, 2009.
[9] Vasly Volkov. Use registers and multiple outputs per thread on GPU. http://www.cs.berkeley.edu/~volkov/volkov10-PMAA.pdf, 1-49, 2010.
[10] Snaider Carrillo, Jakob Siegel and Xiaoming Li. A control-structure splitting optimization for GPGPU. CF '09, 147-150, 2009.

付　　録

```
__device__ int function(int tmp){
#pragma unroll 128
  for(int i = 0;i < LOOP_NUM;i++){
    tmp = 0xffff & (tmp * tmp + tmp);
  }
  return tmp;
}
__global__ void kernel(int *input_array,int *out_array){
  int input_val = input_array[blockIdx.x * blockDim.x + threadIdx.x];
  int num = input_val;
  while(1){
    num = function(num);
    input_val--;
    if(input_val == 0)break;
  }
  out_array[blockIdx.x * blockDim.x + threadIdx.x] = num;
}
__global__ void kernelEnhanced(int *input_array,int *out_array){
  __shared__ int counter;
  if(threadIdx.x == 0){
    counter = BLOCK_DIM;
  }
  __shared__ int result_temp_array[ARRAY_PER_BLOCK];
  __syncthreads();
  int current_array = threadIdx.x;
  int input_val = input_array[blockIdx.x * ARRAY_PER_BLOCK + threadIdx.x];
  int num = input_val;
  while(1){
    num = function(num);
    input_val--;
    if(input_val == 0){
      result_temp_array[current_array] = num;
      int next_array = atomicAdd(&(counter),1);
      if(next_array < ARRAY_PER_BLOCK){
        current_array = next_array;
        input_val = input_array[blockIdx.x * ARRAY_PER_BLOCK + next_array];
        num = input_val;
      }
      else{
        break;
      }
    }
  }
  __syncthreads();
  for(int i = 0;i < ARRAY_PER_BLOCK/BLOCK_DIM;i++){
    out_array[ARRAY_PER_BLOCK * blockIdx.x + threadIdx.x * (ARRAY_PER_BLOCK/BLOCK_DIM) + i]
      = result_temp_array[threadIdx.x * (ARRAY_PER_BLOCK/BLOCK_DIM) + i];
  }
}
```

図 A·1 動的割り付け実験に用いたカーネルコード.

Fig. A·1 The kernel code used for the experiment of the dynamic work assignment method.

```
__device__ int functionA(int tmp){
#pragma unroll 100
  for(int i = 0 ;i < LOOP;i++){
    tmp = (0xffff&((tmp * tmp)+tmp));
  }
  return tmp;
}
__device__ int functionB(int tmp){
#pragma unroll 100
  for(int i = 0 ;i < LOOP;i++){
    tmp = (0xffff&((tmp * tmp)+tmp));
  }
  return tmp;
}
__global__ void kernel(int *length_ary,int *out_ary){
  int length = length_ary[threadIdx.x + blockDim.x * blockIdx.x];
  int data = threadIdx.x + blockDim.x * blockIdx.x;
  if(length_ary[data] & 0x4){
    data = functionA(data);
  }
  else{
    data = functionB(data);
  }
  out_ary[threadIdx.x + blockDim.x * blockIdx.x] = data;
}
__global__ void kernelEnhanced(int *length_ary,int *out_ary){
  int index = DATA_PER_THREAD * (threadIdx.x + blockIdx.x * BLOCK_DIM);
  int counter_A = index,counter_B = index;
  while(true){
    {
      int use_data = -1;
      for(int i = counter_A;i < index + DATA_PER_THREAD;i++){
        if(length_ary[i]&0x4){
          use_data = i;
          counter_A = i+1;
          break;
        }
        counter_A = i + 1;
      }
      if(use_data ≥ 0){
        int data = use_data;
        out_ary[use_data] = functionA(data);
      }
    }
    {
      int use_data = -1;
      for(int i = counter_B;i < index + DATA_PER_THREAD;i++){
        if(!(length_ary[i]&0x4)){
          use_data = i;
          counter_B = i+1;
          break;
        }
        counter_B = i + 1;
      }
      if(use_data ≥ 0){
        int data = use_data;
        out_ary[use_data] = functionB(data);
      }
    }
    if((counter_A ≥ index + DATA_PER_THREAD) && (counter_B ≥ index + DATA_PER_THREAD))break;
  }
}
```

図 A·2 分岐統一化実験に用いたカーネルコード.

Fig. A·2 The kernel code used for the experiment of the branch unification method.