卒業論文

題目

CUDAを用いた大規模データにおける分割手法

指導教員

大野 和彦 講師

2014年

三重大学 工学部 情報工学科
コンピュータソフトウェア研究室

山出　拓矢(410857)

内容梗概

近年, 科学計算や数値解析など様々な分野において大規模かつ高速なコンピュータが求められ, 処理するデータの規模も大規模なものになっている. そのため, CPU よりも性能向上が著しい GPU に汎用計算を行わせる GPGPU の取り組みが活発となっている.

しかし, 処理対象のデータ量が GPU のメモリサイズを上回る場合, 基本的には一度に計算することができない. そこで, データを GPU が処理可能なサイズに分割するわけなのだが, このとき, データ配列を並び替えないで分割し処理する手法と, GPU が処理しやすいよう並び替え, その後同様に分割し処理する手法を実装し, それぞれの実行処理時間を計測することで, どちらがより高速に処理できるのかを検証, 評価を行った.

実装した各手法に対し, $N \times N$ の 2 つの正方行列の行列積の計算時間を計測した. その結果, データを並び替えた場合の方が, 処理プロセスの増加にもかかわらず, 実行処理時間が短くなった.

Abstract

In recent years, large-scale and high-speed computer obtained in various fields such as scientific computing and numerical analysis, the scale of the data to be processed is also made to the large ones. For this reason, general purpose computing to a significant improvement in performance than the CPU GPU GPGPU efforts to perform has become active. However, if the amount of data to be processed exceeds the memory size of the GPU, it can not be basically calculated once. Therefore, although I such not to divide the data to the GPU that can handle the size, this time, the data sequence and a method of dividing processing without rearranging, GPU is manageable as Sort, and then implement a method to split was treated in the same manner, by measuring each of the execution processing time, which can be processed at a higher speed verifying whether, were evaluated. For each implemented method, was measured computation time of the matrix product of two square matrices of $N \times N$. As a result, towards the case of rearranged data, despite the increase in process, the execution time It shortened.

目 次

図 目 次

表 目 次

1　はじめに

1.1　背景

近年、科学計算や数値解析など様々な分野において大規模かつ高速なコンピュータが求められ, 処理するデータの規模も大規模なものになっている. そのため, CPU よりも性能向上が著しい GPU に汎用計算を行わせる GPGPU の取り組みが活発となっている. しかし, 処理データ量に対し GPU のメモリがそれを下回る場合一度に計算することができない可能性がある. そこで, この問題に対し, GPU 側のメモリ容量が少なく, 単体の GPU では一度に処理できない量のデータを処理するための手法について研究している.

1.2　研究目的

現在の CUDA のデータ処理は, GPU 上に処理対象のデータ配列全体と計算結果を格納するデータ領域を確保した上で実行されるのだが, この場合, GPU 上にそれぞれのデータ領域を同時に確保できない場合, データ処理が実行不可能となる. そこで, 処理対象のデータを複数回に分けて転送・処理を行うことで, GPU 側のメモリ容量が劣る場合でもデータ処理が実現可能となる.

1.3 本文構成

本文の構成は以下のようになっている．第 2 章でまず CUDA の概要について述べ, 第 3 章に今回提案した各手法の実装, 第 4 章で性能の評価を行い, 最後に第 5 章でまとめを行う.

2 概要

2.1 CUDA

2.1.1 概要

CUDA(Compute Unified Device Architecture) は, NVIDIA 社が提供する GPU に対する GPGPU を目的とした統合開発環境である. 演算性能の高さと並列処理により, 科学計算など様々な分野で応用されている. また, C 言語を拡張した構文によって記述されるため, C 言語を学習したユーザーなら比較的簡単に記述が可能と考えられる.

2.1.2 データ処理の流れ

CUDA におけるデータ処理は基本的に以下の流れで実行される.

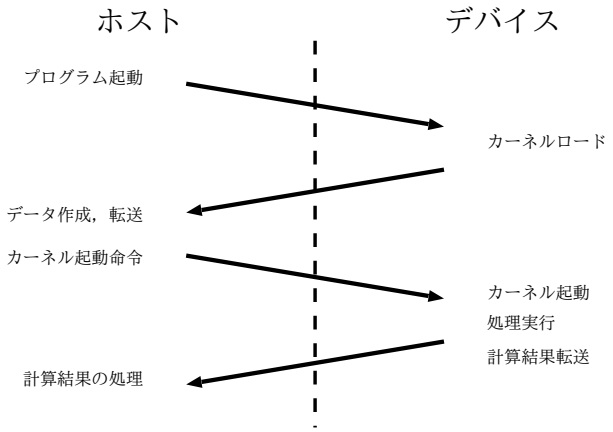

図 2.1: CUDA プログラムの実行の流れ

2.1.3 ブロック・スレッド

CUDA を用いて GPU 上でデータを処理する場合, 同じプログラムが複数のスレッド上で実行されることによってデータを処理する. スレッドの総数が大きくなっても効率よく処理できるように, CUDA ではグリッドとブロックという概念によって階層的に管理される. グリッドとブロックはそれぞれ x 方向, y 方向, z 方向の 3 次元的に配置されるのだが, グリッド内のブロック数は最大 65535 個, ブロック内のスレッド数は最大 1024 個となっている (この最大数はハードウェアによって異なる). これを超えて配置すると実行時にエラーとなる.

2.1.4 カーネル関数

GPU 側で実行させるプログラムのことを指す. C 言語のように引数を指定し, さらに, 実行時に使用するスレッド数 (厳密にはグリッド内のブロック数とブロック内のスレッド数) を指定する必要がある. 関数本体を記述するにあたって, ホスト側とデバイス側のどちらで動作する関数なのかを表す関数修飾子が必要となる. 以下にプログラムの記述例を記す.

記述例

```
__global__ void kernelFunction(int* inA,int* inB,int* inC)
{
  int x = threadIdx.x;
  int y = threadIdx.y;
  int z = threadIdx.z;
      :
      :
}
int main()
{
  dim3 grid(5,5);
  dim3 block(4,4,4);
     :
  kernelFunction<<<4,64>>>(A,B,C);
     :
}
```

注意: 便宜上記述を一部省略しています.

この __global__ の部分が, ホスト側とデバイス側のどちらで動作する関数なのかを表す関数修飾子を指し, これは, GPU で動作する関数であることを宣言している. また, main() 関数で呼ばれたカーネルが実行される時, 各スレッドにインデックス番号が付き, その情報が格納されるのが threadIdx となる. 今回は 3 次元で構成されている場合について記述したが, 実際はスレッドの次元数によって変数の格納のされ方が変わる. この他にも threadIdx の上位変数である blockIdx と呼ばれるグリッド上で動作しているブロックのインデックス番号を格納する変数があり, 配列自体は実質 2 次元で構成されるので, blockIdx.x , blockIdx.y にインデックス番号が格納される.

main() 関数内の kernelFunction の右の <<< >>> は, <<< グリッド内のブロック数, ブロック内のスレッド数 >>> を指しており, 今回は,「4 つのブロックをそれぞれ最大 64 スレッドが起動する」, ということを表している.

そして, main() 関数の最初の dim3 は, CUDA のブロックやスレッドの数を指定するための 3 次元ベクトルの整数型の変数の宣言として使われる. ここでは,「dim3 grid(5,5)」は「グリッドの中に 5x5 のブロックを起動する (grid は変数名)」「dim3 block(4,4,4)」は「ブロックの中に 4x4x4 のスレッド

を起動する (block は変数名)」と言う意味で使用している. この場合, 記述例の kernelFunction<<<4,64>>> を kernelFunction<<<grid,block>>> とすれば, 25 個のブロックに, 最大 64 個ずつスレッドを起動させることになる.

3　実装

3.1　基本的な処理の流れ

データ処理の基本的な流れを図 3.2 に示す. CPU から GPU へのデータ転送では, プログラム上において, cudaMemcpy 関数を for 文や while 文といったループ文の内側で転送元と転送先の引数にオフセット値を指定し, また, 1 回の転送における転送量を指定することで, 複数回に分けて処理対象の配列のデータ処理が実現可能となる.

図 3.2: データ処理の基本的な流れ

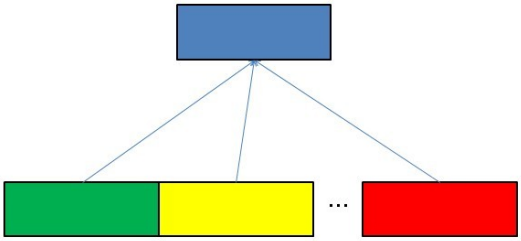

図 3.3: 理想的ななデータ処理の例

3.2 処理対象データ処理

3.2.1 データ構造を並び替えない場合

処理データ配列に対し処理を何も施さずにそのままの状態で処理する場合, データの処理方法によっては, 処理に必要なデータが小さいサイズで複数のデータブロック上に存在することがあり, その場合, 要求データの転送回数と要求データの先頭アドレスの取得のための時間が増え, 結果的に全体の処理時間の増加につながる.

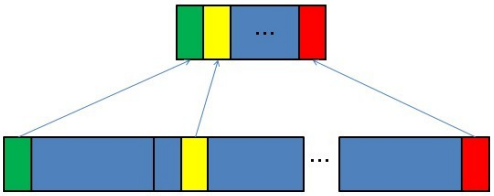

図 3.4: 非効率なデータ処理の一例

3.2.2 データ構造を並び替える場合

処理に必要なデータを可能な限り一度に転送できるようにデータを並び替えることで転送回数および転送データのアクセス時間を少なくし, 最終的にはデータ処理における全体の実行時間を減らすことができる. しかし, この手法を採用する場合, データの並び替えによる処理時間が新たに追加され, このとき, データ量の増加に従って並び替えの時間が長くなるほど, データ配列を並び替えない方が実行時間が短くなる可能性があると考えられる.

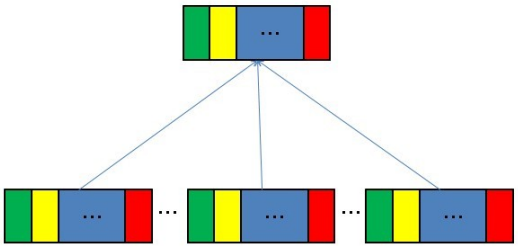

図 3.5: 最適化されたデータの並びによるデータ処理の例

4 性能評価

4.1 評価プログラムの内容

今回は, $N \times N$ の 2 つの正方行列 A,B の行列積を計算し, 計算結果を行列 C に格納する. 実行時における行列のサイズは, $1024(2^{10})$ から開始し, 1 辺のサイズを 8192 まで 2 倍ずつ増やしていく. また, 行列処理に使用したブロック・スレッド数は, 共に 1 ブロックあたりの最大スレッド数である 1024 とする. 各データサイズおよび適用した手法の下で複数回実行し, それらの平均実行時間を実行結果とする. 実行結果は Table4.1 に示す.

4.2 考察

今回の実行結果から, わずかながら変形法による実行時間の短縮を確認することができた. いずれのデータサイズにおいても, その効果は全体

表 4.1: 実行結果 (s)

データサイズ	変形法未適用	変形法適用	変形法適用 (変形時間未加算)
1024 x 1024	0.217	0.212	0.208
2048 x 2048	2.234	2.162	2.120
4096 x 4096	19.155	18.691	18.636
8192 x 8192	207.421	203.923	203.016

の約 3,4%程度という結果となり, 今回使用したプログラムにおいては特に大きな改善策となりうることはなかったと考えられる. データサイズが 8192 における変形法を適応した場合の変形時間を考慮した場合としなかった場合の実行時間の差が他のデータサイズのものと少し異なったものとなっているのは, 実行時間を計測する際, データサイズが大きくなるほど各実行時間のばらつきが大きくなっており, そのため今回においては偶然このような結果になったと考えられる.

4.3 評価環境

以下に今回評価に使用したGPUの性能を記す.

表 4.2: 評価に使用したGPUの性能

デバイス名	GeForce GTX TITAN
グローバルメモリ	6143MB
コア数	2688 個
動作クロック数	876MHz
L2 キャッシュ	1500MB
最大スレッド数	1024
最大ブロック数	65535

5 あとがき

本研究では, 大規模なデータ処理において, 処理対象データの分割に対する手法の提案とその機能の実装, 評価を行った. その結果, わずかではあるものの処理の高速化に成功したが, データサイズの増加に伴って各スレッド間の同期処理における時間も増加していることがわかった.

今後の課題として, GPU 内に搭載されているさらに高速なメモリを利用したり, スレッド間の同期に要する時間を減らすことで, さらなる高速化を試みる必要がある.

謝辞

本研究を行うにあたり, ご指導, ご助言いただきました大野和彦講師に深く感謝いたします. また, 様々な局面にてお世話になりましたコンピュータソフトウェア研究室の皆様にも心より感謝いたします.

参考文献

[1] 青木尊之・額田彰, はじめての CUDA プログラミング, 2009

[2] 岡田賢治, CUDA 高速 GPU プログラミング入門, 2010

A　プログラムリスト

B　評価用データ