6J-1

GPUを用いた処理の自動チューニング

菊地 翔太† 久保田 光一‡

中央大学大学院 理工学研究科 情報工学専攻†‡

要約: 近年, GPU を汎用計算に用いる手法の GPGPU が注目されている. GPU を用いることで高速な並列処理を行うことが可能になり, 処理時間の短縮も可能である. しかし, 処理や問題サイズによっては CPU で実行した方が速い場合もある. 本研究では問題サイズによって GPU 用のライブラリを用いて処理をするか CPU 用のライブラリを用いて処理するかを自動で判断し, 処理を行うライブラリを作成した.

キーワード: GPGPU, CUFFT, FFTW, 自動チューニング

1 背景

GPU を用いて並列処理を行うことで処理時間の短縮が可能なことが知られている. しかし, 入力のデータサイズによっては CPU で実行した方が速い場合もある. また, GPU での実行ではホスト (CPU) 側のメモリとデバイス (GPU) 側のメモリとの間で通信をしなくてはならない. よって, 通信コストを考慮して GPU と CPU を使い分けなくてはならない.

2 目的

本研究の目的は NVIDIA 社の GPGPU 用の統合開発環境 CUDA に付属する FFT ライブラリの CUFFT と CPU 用の FFT ライブラリである FFTW を用いて 1 次元, 2 次元, 3 次元の各 FFT の処理を自動切り替えを行う. すなわち, 問題サイズによって GPU 用のライブラリを用いて処理をするか CPU 用のライブラリを用いて処理するかを自動で判断し, 処理を行うライブラリを作成することである.

3 自動チューニング

自動チューニングとは性能を上げるためにパラメタを自動で設定することや最適なアルゴリズムを選択することをいう. 本研究ではライブラリを用いた処理をする際に CPU と GPU の実行時間の長さによって問題サイズに応じて処理を最適に振り分けることと定義する [1].

4 本研究で用いた FFT ライブラリ

今回は CPU 用の FFT ライブラリとしては FFTW[2], GPU 用の FFT ライブラリとして統合開発環境 CUDA[4] に付属する CUFFT を用いた.

4.1 FFTW

FFTW は離散フーリエ変換を計算する高速な C のルーチン集である. パフォーマンスを最大に発揮するためにハードウェアの条件に合わせたアルゴリズムで実行する. 配列サイズが, 小さい素数で素因数分解される場合に最も効率的に機能する [2][3].

4.2 CUDA

CUDA とは NVIDIA 社の GPU 向けに開発された統合開発環境である. 言語としては C 言語をベースに GPU 用に拡張されたもので CUDA に付属する nvcc コンパイラで実行する [4][5][6].

4.2.1 CUFFT

CUFFT は FFTW と似たインターフェースを持っていて, GPU で簡単かつ高速に実行することができる. GPU で FFT を実装する場合には各 GPU 用に変数のカスタマイズをしなくては GPU の性能が発揮できない. しかし, CUFFT を用いることによってカスタマイズをすることなく, どの環境でも高速に FFT を行えるようになる. また, CUFFT ライブラリ 2.3 から倍精度での計算も可能になっている [4][5][6].

5 実行時間の計測

実行時間の計測は CUDA Utility の関数を使って計測する. CUDA Utility を用いることでどの環境で実行しても同じ方法で実行時間の計測が可能になる. 経過時間はミリ秒で計ることができる.

CUFFT で実行する FFT の実行時間を T_{CUFFT} とし, FFTW で実行する FFT の実行時間を T_{FFTW} とする. また, GPU の場合は FFT の実行時間以外にも転送コストが発生する. ホストからデバイスへのメモリ転送時間を $T_{HostToDevice}$ とし, デバイスからホストへのメモリ転送時間を $T_{DeviceToHost}$ とする. GPU を用いた場合の実行時間 T_{GPU} を式 (1) と定義する.

$$T_{GPU} = T_{HostToDevice} + T_{CUFFT} + T_{DeviceToHost} \quad (1)$$

CPU を用いた場合の実行時間 T_{CPU} を FFT の実行時間のみを測定して式 (2) で定義する.

$$T_{CPU} = T_{FFTW} \quad (2)$$

6 自動切換えライブラリ

今回は 2 のべき乗のサイズを入力サイズとして実行する.

各データサイズに対して実行時間などのデータを取得し, 与えられた実行環境において CPU で実行するか GPU で実行するかの切り分けを自動的に判断する情報を持った配列と関数を持ったヘッダファイルをカレントディレクトリに出力する.

6.1 データの取得

2 のべき乗のサイズに乱数を代入したものを入力とする. CPU は 10 回, GPU は 100 回ずつ実行し平均と標準偏差を求める. これを指定したサイズまで実行し各サイズに対する平均実行時間と標準偏差を求めていく. 同様にフーリエ逆変換も平均実行時間と標準偏差を求める.

6.2 切り替えポイントの検出

平均実行時間と標準偏差を足した値をそのサイズでの実行時間とする.

各サイズにおいて T_{GPU} と T_{CPU} でどちらが実行時間が短いかを記録した配列を *point* とする.

$$point[\log_2 size] = \begin{cases} 1 & (T_{GPU} \geq T_{CPU} \text{のとき}) \\ 0 & (T_{GPU} < T_{CPU} \text{のとき}) \end{cases} \quad (3)$$

Automatic Tuning for Computation with GPGPU

† Shota KIKUCHI, Information and System Engineering Course, Graduate School of Science and Engineering, CHUO University

‡ Koichi KUBOTA, Information and System Engineering Course, Graduate School of Science and Engineering, CHUO University

この式 (3) によって各サイズにおいてどちらが実行時間が短いか判別する．0 から 1, または 1 から 0 に変わるサイズを切り替えポイントと定義する．切り替えポイントを 1 つの変数だけでなく配列にすることによって切り替えポイントを複数持っている場合にも対応できるようにした．

6.3 ヘッダファイルの出力

前節で作成した配列を用いて切り替えポイントを持ったヘッダファイルを出力する．

サイズを入力として与えると切り替えポイントの情報を持った配列を参照し，どちらの実行時間が短いか判断しそちらで FFT を行う関数が含まれている．

6.4 ライブラリの利用方法

出力されたヘッダファイルには以下のような関数が含まれている．以下には単精度の 1 次元 FFT の場合を示す．

- fft_forward_exec (struct data_float *$idata$, struct data_float *$odata$, int $size$)
- fft_backward_exec (struct data_float *$idata$, struct data_float *$odata$, int $size$)

data_float という構造体は単精度の x と単精度の y をメンバーに持つ構造体で，x には実数を代入し y には虚数を代入する．関数に代入する変数は以下のようになっている．

- *$idata$：入力データを格納するポインタの配列
- *$odata$：FFT の結果が代入されるポインタの配列
- $size$：入力データサイズ

$idata$ は $size$ 分のメモリを確保した単精度のポインタの配列を用意し，入力データを代入する．$odata$ も同様に $size$ 分のメモリを確保した単精度のポインタの配列を用意する．上記の関数に入力データの *$idata$, 出力結果が代入される *$odata$, あとは入力のデータサイズ $size$ を渡せば GPU 用のライブラリで実行するか CPU 用のライブラリで実行するか自動で判断し，実行する．結果は *$odata$ に代入される．また，data_float は CUFFT で実行する場合はそのまま実行できるが，FFTW で実行する場合には型変換を行う必要がある．その型変換も関数の方で自動で行うようにしている．

7 実験

以下の環境にて実験を行った．

OS	Ubuntu10.04
CPU	Intel Corei7 870 2.93GHz
memory	4GB
GPU	NVIDIA GeForce GTX 480
Driver Version	270.41.19
CUDA Version	4.0

CPU は 10 回, GPU は 100 回ずつ各サイズごとに実行時間を測定してプロットすると図 1 のようになる．横軸の入力データサイズは底が 2 の対数を取っている．また，縦軸の実行時間は底が 10 の対数を取っている．

次の図 2 は図 1 の各サイズごとに平均実行時間と標準偏差を求め，足したものになる．

図 2 の結果を用いて CPU と GPU どちらが実行時間が短いかの情報を持った配列を作成する．この実験環境の場合はデータサイズが 2^{12} までは CPU のライブラリを用いて実行し，2^{13} より大きいデータサイズのものは GPU のライブラリを用いて実行するようなライブラリを出力することができた．

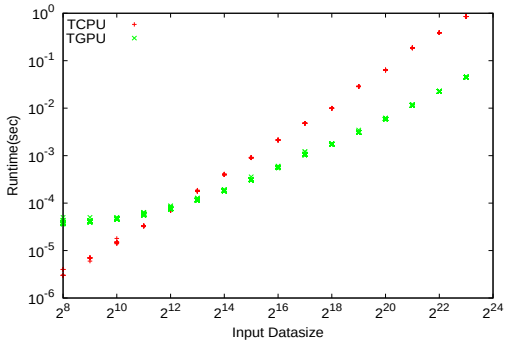

図 1　データサイズごとの実行速度

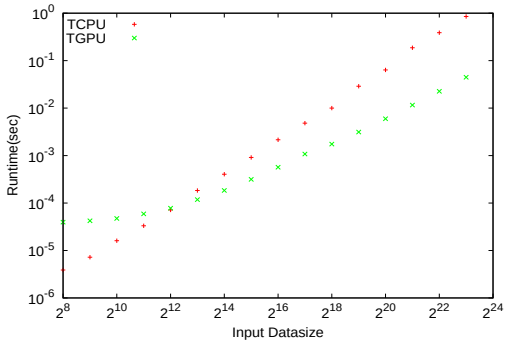

図 2　データサイズごとの実行速度

また，同様の流れで CPU に Core 2 Duo, GPU に GeForce 8800 GTX を用いた実験機と CPU に Core 2 Duo, GPU に GeForce 320M を用いた実験機でも自動で切り分けを行うライブラリの出力に成功した．

8 まとめと今後の課題

CPU と GPU の FFT ライブラリを用いた処理において，入力データのサイズによる切り替えポイントの検出プログラムの作成およびその情報を持った切り替えポイントのライブラリ出力プログラムを作成した．各実験環境において切り替えができてることを確認した．

今回は FFT ライブラリを対象としてプログラムを作成したが BLAS ライブラリへの対応，また NVIDIA 社の GPU だけでなく AMD 社の GPU にも対応できるように Open CL などを用いたプログラムなどが課題として挙げられる．

参考文献

[1] 片桐 孝洋, **“ソフトウェア自動チューニング”**, 慧文社, 2004.
[2] **“FFTW”**, http://www.fftw.org/, 最終アクセス日 2012 年 1 月 13 日.
[3] **“FFTW@wiki”**, http://www32.atwiki.jp/amaeda/, 最終アクセス日 2012 年 1 月 13 日.
[4] **“NVIDIA Developer Zone”**, http://developer.nvidia.com/category/zone/cuda-zone, 最終アクセス日 2012 年 1 月 13 日.
[5] 青木 尊之, 額田 彰, **“はじめての CUDA プログラミング”**, 工学者, 2009.
[6] **“OpenGL de プログラミング”**, http://wiki.livedoor.jp/mikk_ni3_92/, 最終アクセス日 2012 年 1 月 13 日.