3次多項式パラメタ推定計算の CUDAを用いた実装

（CUDAプログラミングの練習として）

Estimating the Parameters of 3rd-order-Polynomial with CUDA

ISS

09/11/12

問題の選択

目的

- CUDAプログラミングを経験（試行錯誤と習得）
- 実際にCPUのみの場合と比べて高速化されることを体験

問題（インプリメントする内容）

滑降シンプレックス法（Nelder's Downhill Simplex Method）による3次関数フィッティング

- 目的関数値評価回数が多い（今回のテスト対象としては好都合）
- インプリメントが容易（不要な個所で悩まずに済む）
- 手法自体（の振る舞い）が理解しやすい（動作確認が容易）

- データ点毎に完全に独立した残差計算の存在（並列化対象）
- 今後，当てはめ問題をCUDAを用いて実装する予定がある（ノウハウの獲得）
- 「3次」なのは特に理由なし

問題の概要

3次関数フィッティング：問題の作成

- 3次多項式 $y = f(x;\mathbf{p}) = p_3x^3 + p_2x^2 + p_1x + p_0$

  の正解パラメタ$\mathbf{p}_t$を適当な値に決める

- 2次元平面上にn個のデータ点 $(x_i, y_i = f(x_i;\mathbf{p}_t)) \quad i = [0, n-1]$

  を適当に用意する

- 初期パラメタ値$\mathbf{p}_0$を適当（$\mathbf{p}_t$から遠すぎず近すぎず）に与える
  - 初期Simplexの残りの頂点は$\mathbf{p}_0$の周りに適当に配置

- 最小二乗法

  minimize $R(\mathbf{p}) = \sum_{i=0}^{n-1} \left(f(x_i;\mathbf{p}) - y_i \right)^2$

  として$\mathbf{p}$を推定する

※滑降シンプレックス法自体のパラメタ（反射や収縮時の倍率値）は，``Numerical Recipies in C”掲載コードと同一の値を用いた．これは，Nelderらが論文内で「経験的に最も良いパラメタ値の組み合わせ」と述べていた値だったと思う．

計算条件

- 浮動小数点数は単精度(32-bit float)
- データ点数n=30000
- 真のパラメタ値 $\mathbf{p}_t=(p_0, p_1, p_2, p_3)$ = (0.0, 0.8, -0.08, 0.0014)
- 初期パラメタ値 $\mathbf{p}_0=(p_0, p_1, p_2, p_3)$ = (0.5, 0.7, 0.0, 0.0)
- 初期シンプレックス生成用 $\lambda=(\lambda_0, \lambda_1, \lambda_2, \lambda_3)$ =(0.1, 0.1, 0.1, 0.01)

  初期シンプレックスの頂点の一つを$\mathbf{p}_0$として,
  残りの4頂点（0番~3番 と呼ぶことにする）は以下のように作る.
  j番の頂点は，$\mathbf{p}_0$から1つのパラメタだけを動かした場所

$$j\text{番頂点の}p_i = \begin{cases} \mathbf{p}_0\text{の}p_i & if\ (i \neq j) \\ \mathbf{p}_0\text{の}p_i + \lambda_i & other \end{cases}$$

として作る.

<u>計算時間比較時：</u>

floating point演算の結果が完全に一致しないので,
何らかの収束条件で計算を止めるのでなく,
滑降Simplex法の手続きを一定回数(200回)行った時点で止めることにした.

※各回で選択される手続きの種類によって関数値の計算回数は異なるので,
浮動小数演算結果の違いにより手続き選択に差異を生じた場合,
この方法では本当に厳密な時間比較とはならない.
ただ，今回比較した4種のプログラムの選択手続きが200回の間で「ほとんど」同じである
ことは事前確認した.
（例えば，C++_CPUとC++_CUDAでは195回目までは同一の選択がなされていた）

コードフロー概要

※青字はCUDA使用版プログラムのみ

- device選択(CUDAランタイム初期化)
- データ点群作成，初期シンプレックス作成等
- device memory, host側メモリはここで確保
- データ点群はここでdevice memoryへ転送しておく

200回　滑降Simplex法の手続きループ

ここの時間を計測

- メモリ解放等の後始末処理
- 結果出力表示等

device上で計算する部分

CUDAを使うのは目的関数計算の個所

- CPUのみ版のプログラムでは，目的関数

$$R(\mathbf{p}) = \sum_{i=0}^{n-1} \left(f(x_i;\mathbf{p}) - y_i \right)^2$$

を素直にループ計算する.

- CUDA使用版のプログラムでは…

$$R(\mathbf{p}) = \sum_{\text{block個数分}} \sum_{\text{blockサイズ分}} \left(f(x_i;\mathbf{p}) - y_i \right)^2$$

1threadが1データ点の残差2乗値を計算

外側のΣはhostで行う

このΣはblock内の先頭threadが行う

のようにした.
blockサイズ(threads/block)は64である.

kernelコード

```
//※shared memoryの動的割当サイズに （ sizeof(FLOAT_TYPE)*block個数 ） を指定すること！
template< unsigned int N_PARAM, class FLOAT_TYPE >  //N_PARAMはパラメタ個数で，今回は4である．FLOAT_TYPEは今回はfloatである．
__global__ void
CTest_Kernel2(  FLOAT_TYPE* g_in_Xs,              //データ点X座標の配列
                FLOAT_TYPE* g_in_Ys,              //データ点Y座標の配列
                FLOAT_TYPE* g_in_Params,          //パラメタベクトル
                unsigned int g_in_NumOfData,      //データ点個数．＝データ点配列サイズ．
                FLOAT_TYPE* g_out_SumResidues_per_block //各ブロックでの残差２乗値を格納するための配列領域(block個数分のサイズが確保されていること)
                )
{
    //ブロックの最初のN_PARAM個のthreadがshared memoryにパラメタ値をコピーする
    __shared__ FLOAT_TYPE P[N_PARAM];

    if( threadIdx.x < N_PARAM )
    {    P[threadIdx.x] = g_in_Params[threadIdx.x];    }
    __syncthreads();  //パラメタ値のコピー完了を待つための同期

    //1threadが1データ点の残差値を計算して，結果格納域 DY[] に入れる
    extern __shared__ FLOAT_TYPE DY[];
    DY[ threadIdx.x ] = 0;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;     //このthreadが扱うデータ点のindexを計算
    if( i < g_in_NumOfData )
    {
        FLOAT_TYPE X = g_in_Xs[i];
        FLOAT_TYPE Y = ((P[3]*X + P[2])*X + P[1])*X + P[0];     //三次関数値 Y = f(x;P) を計算
        FLOAT_TYPE dy = Y - g_in_Ys[i];         //データ点の実際のY座標との差を取り…
        DY[threadIdx.x] = dy * dy;              //その２乗値を結果とする
    }
    __syncthreads();  //計算待ち同期

    //thread群のうちの一人が結果の総和を取り、global memory に書き込む
    if( threadIdx.x == 0 )
    {
        FLOAT_TYPE Sum=0;
        for( unsigned int j=0; j<blockDim.x; j++ )
        {    Sum += DY[j];    }
        g_out_SumResidues_per_block[blockIdx.x] = Sum;
    }
}
```

host側は，g_out_SumResidues_per_block[]を
hostメモリに転送し，要素の総和を取って目的関数値とする．

計算時間計測

C++(CPUのみ), C++(CUDA利用), python(CPUのみ), python(PyCuda利用)
の4種類のプログラムの最適化計算ループ部分に費やす時間を計測.

- 4種のプログラムは同一のデータ点群座標をテキストファイルから読込んで使用
- C++(CPUのみ)は, 計算にCUDAを使わないが, ソースを.cuとして
  C++(CUDA利用)と同条件でビルドした

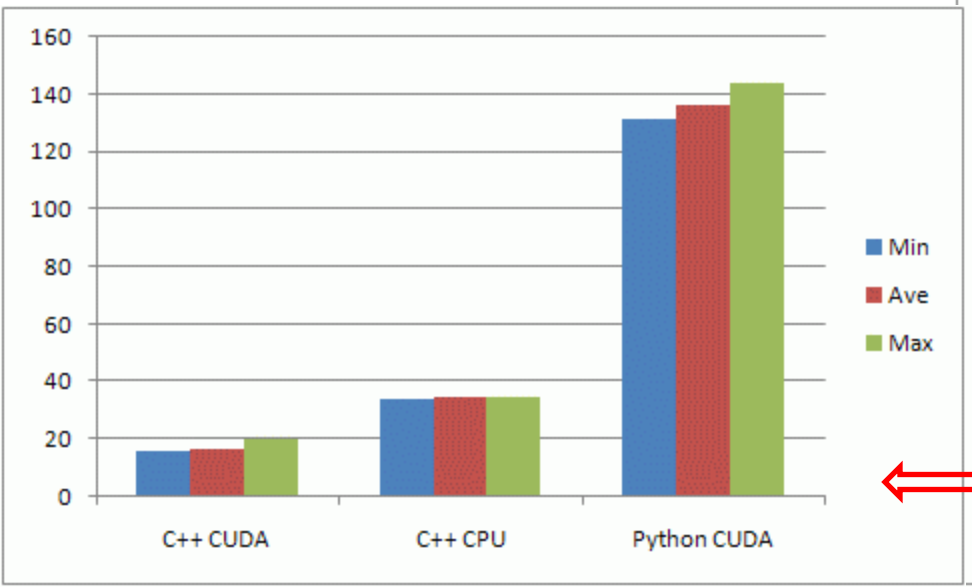
- 使用ハードウェア構成

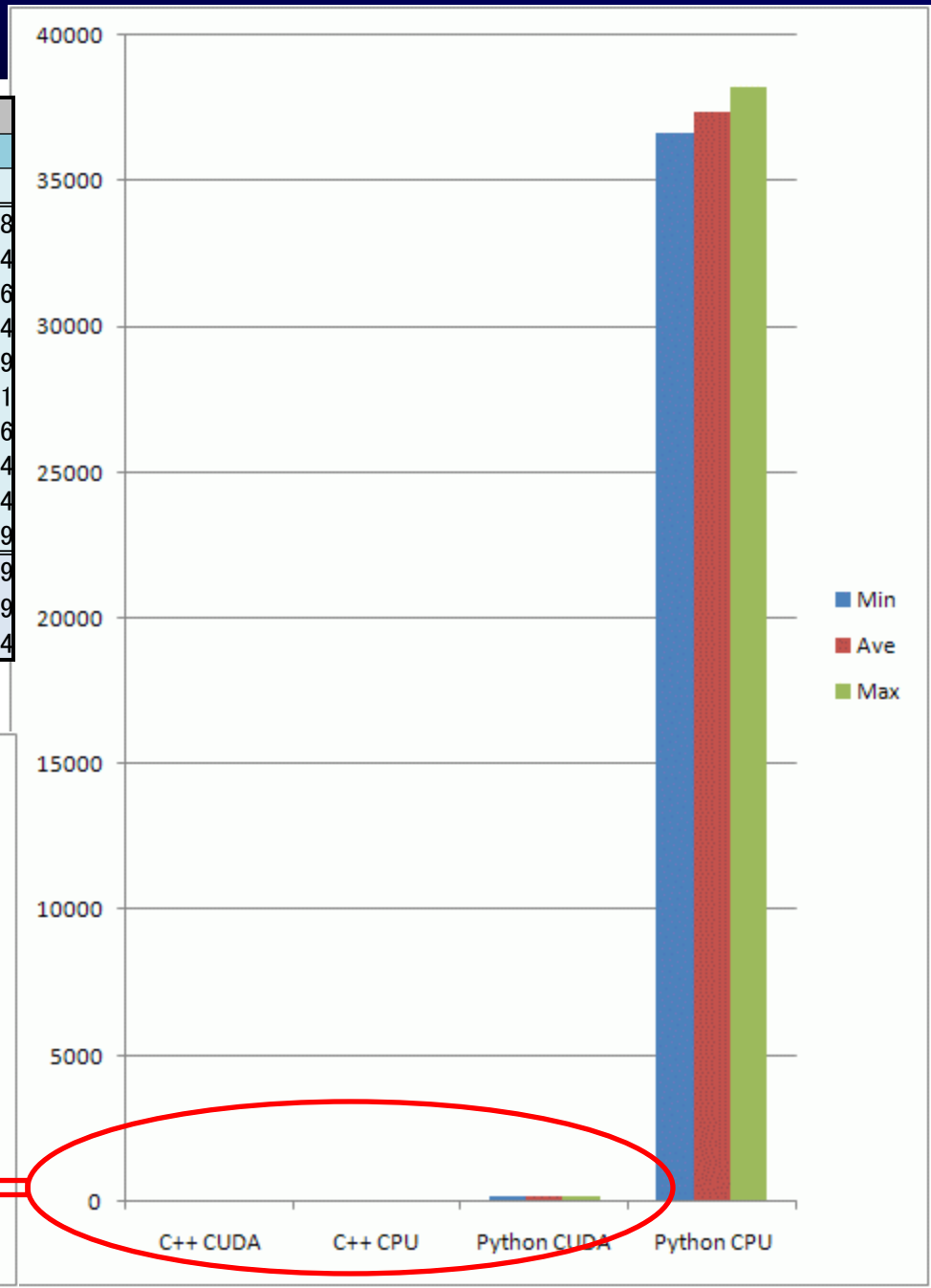
CPU	:	AMD Phenom II X4 945 (3.0GHz)
Memory	:	TRJ JM800QLU-2G (DDR2 PC2-6400) ×4
CUDA-enable device	:	組込用ELSA ETS1060-C4EBB(TeslaC1060 4G GDR3) ×4 （※今回は単一deviceのみ使用）

- Linux(CentOS)上で実行

計算時間計測結果

	プログラム毎の計算時間[ms]			
	C++		Python	
	C++ CUDA	C++ CPU	Python CUDA	Python CPU
1回目	19.95	34.614	136.2609863	36965.778
2回目	16.097	34.73	132.481	37252.154
3回目	16.487	34.336	140.856	37282.06
4回目	17.119	34.434	132.442	38195.374
5回目	16.155	34.879	133.597	37133.879
6回目	16.064	34.517	143.884	37073.071
7回目	16.556	34.365	138.612	36887.006
8回目	16.072	34.421	139.353	38133.234
9回目	16.304	34.636	131.373	38003.304
10回目	16.492	34.54	132.695	36639.149
Min	16.064	34.336	131.373	36639.149
Ave	16.7296	34.5472	136.1553986	37356.5009
Max	19.95	34.879	143.884	38195.374





考察etc

C++

CPUのみの場合に比べてCUDAを用いることで高速化されたが，２倍程度である.

扱っている問題が，「低コスト×大量個数」であったため，メモリ転送量が多いことがボトルネックとなっているのかもしれない.
→「高コスト×複数（<<大量）」のような処理ではより高速化される？

python

CPUのみの場合に比べてPyCuda利用側はかなり速くなっているのだが…

「CUDA利用」以前に，単純に高コスト部分がコンパイルされた(?)ことが大きい可能性もある？

課題

- 今回のコードの書き方によるパフォーマンスの良さがどの程度かについては不明である（計算が正しく動作する，というレベル）.
  良い作法に従うコードならばパフォーマンスが向上すると思われる.

- 複数deviceを同時利用する場合も検討すべきである.