# 2GPU による Cubic セミ・ラグランジュ法の高速化

須藤 郁弥　坂内 恒介　本田 耕一　松田 健護　篠原 歩
東北大学 大学院情報科学研究科

## 1　はじめに

GPU Challenge 2010[1] の規定課題は，Cubic セミ・ラグランジュ法による流体の移流計算であった．空間は $n_x \times n_y$ の直交格子によって表され，$n_x$，$n_y$ は 64 の倍数かつ $2^{15}$ 以下であり，$n_x$ と $n_y$ の積は $2^{27}$ 以下と定められていた．審査では，この条件をみたす複数の問題に対して定められたタイムステップ数 $n_t$ の計算速度が競われた．本稿では，コンテスト向けに開発したプログラムで行った高速化について報告を行う．

## 2　求解アルゴリズム

本節では移流計算に用いた Cubic セミ・ラグランジュ法について説明する．

### 2.1　Cubic セミ・ラグランジュ法

1 次元の移流計算において，ある時刻 $t_{n+1}$ における位置 $x$ の値 $f^{n+1}(x)$ は，速度を $u$，時間間隔を $\Delta t$ とすると時刻 $t_n = t_{n+1} - \Delta t$ の値を用いて，

$$f^{n+1}(x) = f^n(x - u\Delta t)$$

として求められる．よって，時刻 $t_{n+1}$ における $f$ の値は，時刻 $t_n$ における格子点の $f$ の値を補間する事で得られる．格子間隔を $\Delta x$ とし，$x_i = i\Delta x$，$du = -u\frac{\Delta t}{\Delta x}$，$f_i^n = f^n(x_i)$ とすると，Cubic セミ・ラグランジュ法では以下の式により $f_i^{n+1}$ の値を求める．

$$f_i^{n+1} = \begin{cases} c_{3p}du^3 + c_2du^2 + c_{1p}du + c_0 & (u \geq 0) \\ c_{3m}du^3 + c_2du^2 + c_{1m}du + c_0 & (u < 0) \end{cases}$$

ただし，

$$\begin{aligned} c_{3p} &= \tfrac{1}{6}f_{i+1}^n - \tfrac{1}{2}f_i^n + \tfrac{1}{2}f_{i-1}^n - \tfrac{1}{6}f_{i-2}^n \\ c_{3m} &= -\tfrac{1}{6}f_{i-1}^n + \tfrac{1}{2}f_i^n - \tfrac{1}{2}f_{i+1}^n + \tfrac{1}{6}f_{i+2}^n \\ c_2 &= \tfrac{1}{2}f_{i+1}^n - f_i^n + \tfrac{1}{2}f_{i-1}^n \\ c_{1p} &= \tfrac{1}{3}f_{i+1}^n + \tfrac{1}{2}f_i^n - f_{i-1}^n + \tfrac{1}{6}f_{i-2}^n \\ c_{1m} &= -\tfrac{1}{3}f_{i-1}^n - \tfrac{1}{2}f_i^n + f_{i+1}^n - \tfrac{1}{6}f_{i+2}^n \\ c_0 &= f_i^n \end{aligned}$$

### 2.2　2 次元の移流計算

2 次元の移流計算は，1 次元の計算を繰り返し用いることで計算できる．我々は，$x$ 方向の計算 5 回，$y$ 方向の計算 1 回による計算を行った．位置 $(x, y)$ における $x$ 方向の速度を $u$，$y$ 方向の速度を $v$ とし，$x$ 方向の格子間隔を $\Delta x$，$y$ 方向の格子間隔を $\Delta y$ とする．このとき，

$$f^{n+1}(x, y) = f^n(x - u\Delta t, y - v\Delta t)$$

は，まず 5 行に対して $x$ 方向の補間を行う事で，$x$ 座標 $x - u\Delta t$，$y$ 座標 $y - 2\Delta y$，$y - \Delta y$，$y$，$y + \Delta y$，$y + 2\Delta y$ の 5 つの点における値を求め，その値を利用した $y$ 方向の補間によって求められる．

この計算においては，ある点の計算に最大で 2 つ隣の点の値が必要となるが，コンテストでは境界条件として領域の端 2 マスの値は初期値のまま不変とされた．

## 3　1GPU による高速化

本節では，1GPU による計算の実装および行った高速化について述べる．

### 3.1　ブロック化

我々は格子空間を $64 \times 64$ のブロックに分割した．各ブロックでは 64 個のスレッドを作り，各スレッドはブロック内で $y$ 方向の 1 列分の計算を行うようにした．このとき，$f$ の値はブロック内のスレッドが共有して用いるためシェアードメモリに読み込み，グローバルメモリへのアクセスを削減した．また，各行における $x$ 方向の補間で係数部分は使いまわす事が出来るため，計算で用いる 5 行分の係数を保存するようにした．

### 3.2　式変形

Cubic セミ・ラグランジュ法の式において，

$$c_{3p} - c_{3m} = -(c_{1p} - c_{1m})$$

が成り立つことから，$f_i^{n+1}$ の $u \geq 0$ の場合の値を $fp$，$u < 0$ の場合の値を $fm$ としたとき，以下の式が成り立つ．

$$fm = fp - dif \cdot u(u^2 - 1)$$

ただし，$dif = c_{3p} - c_{3m}$ である．この関係を利用する事で，保存する係数を 1 行あたり 1 個減らす事が出来る．さらに，

$$f_i^{n+1} = fp - dif \cdot u(u^2 - 1)(u < 0)$$

として計算を行う事で計算式が簡単になり，条件分岐を削減出来るため，高速化につながった．ここで，$u < 0$ は真のとき 1，偽のとき 0 を返す．

### 3.3 その他の高速化

ループアンローリングにより 1 度の反復で 4 行の計算を行うようにした．また，コンパイル時に生成される ptx ファイルを参考に細かいチューニングを行った．

## 4 2GPU による高速化

コンテストでは 2 個の GPU を使う事が出来た．本節では 2GPU を使う場合に行った高速化について述べる．

### 4.1 計算領域の分割

データはメモリ上で行方向に連続に格納されているため，計算領域は行方向に分割した．このとき，各 GPU が担当する領域の $y$ 方向のサイズは，64 の倍数になるようにそれぞれ $mid = 64\lfloor n_y/128 \rfloor$，$n_y - mid$ とした．また，境界条件から領域の端 2 行は計算不要であるため，各 GPU の担当範囲をそれぞれ第 3 行から第 $(mid + 2)$ 行，第 $(mid - 1)$ 行から第 $(n_y - 2)$ 行とする事で，$y$ 方向の境界条件に関する分岐を削除した．以降，領域の上側を担当する GPU を GPU1，下側を担当する GPU を GPU2 と呼ぶ．

### 4.2 ストリーム処理による同期

2GPU を用いる場合は，正しい計算結果を保証するために GPU 間の通信が必要となる．具体的には各 GPU が担当する領域の外側 2 行について，その値を相手から受け取る必要があり，GPU1 は第 $(mid-3)$ 行，第 $(mid-2)$ 行の情報を GPU2 に，GPU2 は第 $(mid+3)$ 行，第 $(mid+4)$ 行の情報を GPU1 に送る．素朴には，図 1 に示すように各ステップの計算が終了するごとに，CPU を介して必要となるデータの交換を行う．

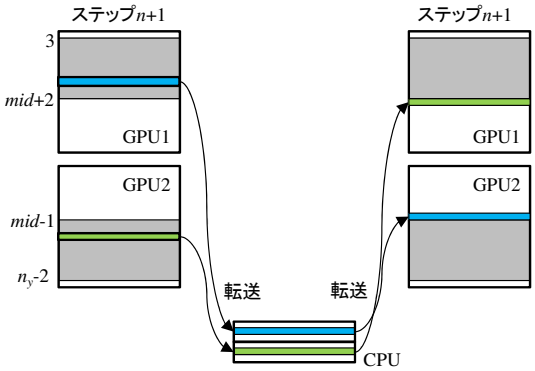


図 1: 通常の同期方法

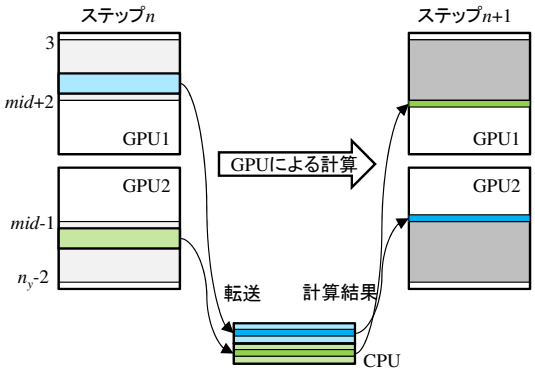


図 2: ストリーム処理を用いた同期

我々は同期に必要なデータ転送をカーネル処理と並列に実行する事で，各ステップの計算後に生じる通信時間を隠蔽した．CUDA では，ストリーム処理により GPU カーネルの実行と GPU-CPU 間の通信は並列に行う事ができる．また，GPU による処理中は CPU による計算も可能である．この性質を利用して，同期処理は次のように行った．

図 2 に示すように，GPU1 が第 $(mid-5)$ 行から第 $mid$ 行の，GPU2 が第 $(mid+1)$ 行から第 $(mid+6)$ 行のデータを CPU に送る．このとき，CPU は送られたデータから各 GPU が必要とする領域について次ステップの値を計算できる．さらに，各 GPU が結果を書き込む領域と，CPU から GPU へデータ転送を行う領域は異なるため，いずれの処理もカーネルの実行と並列に実行できる．

この同期方法は，サイズが小さい問題では CPU の処理やデータ転送がネックとなって通常の同期方法に比べて性

表 1: 実行結果

| 問題 | $n_x$ | $n_y$ | $n_t$ | 1GPU | 2GPU |
|---|---|---|---|---|---|
| 1 | 2,048 | 2,048 | 20,480 | 24.409 | 12.029 |
| 2 | 4,096 | 4,096 | 10,240 | 46.841 | 22.577 |
| 3 | 4,096 | 4,096 | 10,240 | 48.776 | 23.545 |
| 4 | 16,384 | 4,096 | 4,096 | 73.327 | 34.563 |
| 5 | 8,192 | 16,384 | 2,048 | 72.820 | 34.208 |

実行時間は sec.

能が低下したが，サイズの大きい問題では高速化の効果が見られた．プログラムは，$n_y \geq 2^{10}$ かつ $n_x \cdot n_y \geq 2^{21}$ を満たす問題の場合にストリーム処理を用いて同期を行う仕様とした．

## 5 結果

コンテストの審査に使われた 5 問に対する，1GPU を用いた場合および 2GPU を用いた場合それぞれの実行時間は表 1 の通りであった．2GPU による処理では，全ての問題でストリーム処理を用いた同期方法が行われた．1GPU を用いた場合と 2GPU を用いた場合の実行時間を比較すると，2GPU を用いる事で実行速度が約 2.1 倍になっている事が分かる．この事から我々が行った 2GPU を利用した高速化は有効であったと言える．また，実行委員会によるサンプルプログラムで制約上の最大サイズとなる問題 5 を実行すると 612 秒かかるため，2GPU を用いた場合では約 18 倍の高速化となった．

## 参考文献

[1] GPU Challenge 2010 実行委員会．GPU Challenge 2010 http://www.hpcc.jp/sacsis/2010/gpu/