# GPU を用いた直交格子法による流体の移流計算の高速化

福　澤　　太 †,††

## 1. はじめに

GPU Challenge 2010[1] の規定課題部門では，Cubic セミ・ラグランジュ法による流体の移流計算を行う CUDA[2] を用いたサンプルプログラムの高速化という課題が出題された．

本稿では，コンテストにおいて 2 位の評価を受けたプログラムに用いた最適化技法を述べる．また，コンテスト終了後に発見したプログラムの改善点について述べる．

## 2. 高速化の手法

### 2.1 ループアンローリングと式の変形

1 ステップの移流計算を行うカーネル関数のブロック内のスレッド数を $x$ 方向に 64，$y$ 方向に 1 と設定し，1 つのスレッドにおいて，連続する複数の $y$ 方向の格子点を計算するループアンローリングを行った．アンロール回数は，4 回，8 回，16 回，32 回の 4 通りを実装した．

また，1 方向の補間計算を行うサンプルプログラムの gcubic 関数は，速度ベクトルの向きによる分岐が発生しない場合に 22 回の浮動小数点の四則演算を行う．アンローリングを行っていることから，$x$ 方向の補間計算では，連続する $y$ 方向の計算において，速度ベクトルに依存しない部分の演算を使い回すことができる．gcubic 関数内の式を変形し，速度ベクトルに依存しない部分の演算を 14 回の浮動小数点の四則演算で行い，計算結果の値を 5 つのレジスタに保持することで，速度ベクトルに依存する部分の演算を 8 回の浮動小数点の四則演算で行った．この式変形より，浮動小数点演算回数を約 2 分の 1 にできた．

### 2.2 共有メモリの利用

グローバルメモリの読み込みを共有メモリでバッファリングし，一方向の補間計算に必要な五つの格子点の値を共有メモリから読み込むことで，グローバルメモリへのアクセス回数を削減した．

グローバルメモリの読み込み処理は，対象の格子点の一つ前の格子点の処理の間で行い，格子点の処理ごとに__syncthread() を呼び出すようにした．__syncthrad() の挿入は，同一ブロック内のスレッド間の共有メモリの同期に必要であると共に，演算とグローバルメモリへのアクセス処理を交互に記述している中で，適切な位置に__syncthread() を入れることで，TLB ヒット率の向上によると推測される高速化が得られた．

また，各ブロックの x 方向の端の 4 つの格子点の値の読み込み処理は，連続する 4 つの $y$ 方向分を 1 回の読み込み処理で読み込むようにした．

### 2.3 if 文の除去

サンプルプログラムは，if 文が二カ所現れる．

gpu_kernel 関数内の if 文は，境界部分に関して値を変化させないための分岐処理である．対応する位置の速度ベクトルの値を 0 にすることで，計算結果を 1 ステップ前と同じ値にできる．

gcubic 関数内の if 文は，速度ベクトルの向きによって計算式を変更するための分岐処理である．速度ベクトルの絶対値をとることにより，分岐している二つの式は用いる変数を除き係数が一致する．計算に必要な変数の代入を三項演算子を用いて実装することにより，selp.f32 命令として処理を実行することができる．1 格子点あたり 11 回の三項演算子を使い実装した．

### 2.4 アクティブブロック数の最大化

オンレジスタによる計算回数の削減には，多くのレジスタを必要とし，初期の実装ではアクティブブロック数が 3 であった．

オンレジスタによる処理を最大限に生かすために，次のようなコードの修正を行いアクティブブロック数をできるかぎり多くした．

- 命令順序の入れ替えによるレジスタ数の削減
- 生存期間の長い変数をできる限り排除し，変数を局所的に使うようにコードを修正
- __syncthread() を挿入する位置をレジスタ数が最大となる箇所に調整し，コンパイラの最適化を抑

† 東京工業大学 大学院情報理工学研究科 計算工学専攻
†† 現在，株式会社フィックスターズ

制することによるレジスタ数の削減

- 1 ブロックあたりの共有メモリを，長さが $72 \times 8$ の float 配列として実装

提出したプログラムにおいて，1 スレッドあたり 40 レジスタ（アンロール数が 32 の場合は 41 レジスタ），1 ブロックあたり $2344 + 16$ バイトの共有メモリを使っている．Tesla C 1060 ではアクティブブロック数が 6（アンロール数が 32 の場合はアクティブブロック数が 5）となった．

### 2.5 グローバルメモリのアライメントの取れた位置から読み込み

グローバルメモリへのアクセスは，アライメントが取れていない位置よりも，アライメントが取れている位置に対する処理が高速である．

速度ベクトルの配列 u と v の各値が格納されている位置を，$x$ 方向に 2 増やした位置に移す初期化処理を行った上で，各ステップの計算を行った．この初期化処理により，読み込み処理は，全て 64 バイトアライメントが取れた位置に対して行う．しかし，書き込み処理は，常に 64 バイトアライメントから 8 バイトずれた位置に対して行う．

### 2.6 アドレス計算の簡素化

グローバルメモリに対する読み書き処理を行うためのアドレス計算に関して，次のような整数演算の簡素化を行った．

- 各スレッドの最初の格子点に関するアドレス計算について，共通化可能な演算の共通化
- 各スレッドの二番目以降の各格子点のアドレス計算を，$y$ 方向で一つ上のアドレスと $x$ 方向の格子点数である $nx$ の和で計算
- 乗算と除算の演算に関して，ビット演算に置き換え可能な演算を置き換え
- 入力の制限から，__mul24 関数に置き換え可能な演算を置き換え

### 2.7 配列 u と v を float2 配列に変換

速度ベクトルの配列 $u$ と $v$ の同一位置の値をグローバルメモリ上に連続して並べる初期化処理を行った．これにより，float2 の値として 1 回の読み込みで速度ベクトルを読み込むことができ，グローバルメモリへのアクセス回数を削減できた．

### 2.8 2GPU 処理の実装

2 つの GPU を用いて処理を行うために，格子を上下二つに分割し，2 つの GPU でそれぞれ上下の格子の計算を行った．

同期処理は，バリア同期を用いて行った．上下の境界部分の格子点は値が変化しないことを用いて，同期を 2 ステップに 1 回行った．

### 2.9 格子点数による処理の切り替え

様々な格子点数のテストデータを作成し，各アンロール数ごとの実行時間計測した．計測結果から，格子点数によって利用する GPU の数やアンロール数を切り替えて処理を行うようにした．

しかし，審査に用いられたデータの格子点数は全て巨大なものであったため，小さいデータを対象とした 1GPU によるアンロール数 8 や 4 の場合の処理は使われることが無く，2 つの GPU を用いたアンロール数 16 のアルゴリズムのみが評価に用いられた．

## 3. まとめと今後の課題

ループアンローリングとオンレジスタによる浮動小数点演算の削減及びメモリアクセスの最適化等の方法を用いて移流計算の高速化を行った．

しかし，以下に述べる改善等によって，コンテスト終了後にさらなる高速化が達成できた．また，以下の改善によりレジスタ数に余裕ができたことから，コンテスト中にはアクティブブロック数を 6 にすることができなかった 32 アンロールや 64 アンロールにおいて，16 アンロールを超える高速化が達成できると推測される．

### 3.1 テクスチャメモリの利用

グローバルメモリの読み込みを，テクスチャメモリに変えることで，アライメントが取れていない位置の読み込みを，アライメントの取れている位置とほぼ同等の実行時間で実現できる．読み込みをアライメントが取れていない位置にすることで，書き込みをアライメントのとれた位置にすることができ，提出したプログラムから約 6%の高速化が達成できた．

### 3.2 fabs 関数の利用

最初のステップの計算を始める前に初期化処理として，速度ベクトルの配列 u と v の各値の絶対値を計算し，各スレッドは処理のはじめに符号を表すビット値を押し込んだ値を読み込んでいる．

この処理を fabs 関数を用いて実装することにより，初期化処理が不要になると共に，符号を表すビット値の読み込みが必要なくなる．これにより，約 1%の高速化が達成できた．

## 参 考 文 献


1) GPU Challenge 2010 実行委員会. GPU Challenge 2010. http://www.hpcc.jp/sacsis/2010/gpu/.
2) NVIDIA Corporation. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.