Vol.2010-HPC-124 No.1
2010/2/22

GPU を使用した陰関数グラフ描画の高速処理法

鈴 木 省 吾[†1] 村 尾 裕 一[†2]

本稿では，陰関数グラフ描画における描画点判定の処理を，GPU で並列処理を行う方法について述べる．一般に陰関数の描画点を正確に求めることは難しいが，全描画セルについて判定するという方法を採り，その計算に区間演算とモジュラー算法を適用する．この手法による実装を，GPU の統合開発環境である CUDA を用いて行い，GPU 実装時のデータ構造や算法についての指針を示し，具体的な並列化の手法を述べる．実験の結果，この計算が並列処理によく適合することが確認されたが，さらにメモリアクセスのレイテンシ隠蔽のため適切な占有率を保つことにより，最大約 20% の速度向上を得た．また，区間数の冪乗計算がボトルネックとなっていることが判明したが，これは数理的な解析が必要であることを指摘しておく．

An Efficient Method for Drawing Implicit Functions Exactly Using GPU

Shogo Suzuki [†1] and Hirokazu Murao[†2]

This paper describes a method to discriminate the plotted points of implicit functions by making use of parallel computation on GPU. In general it is difficult to obtain an exact set of points, and we employ such a method to perform on/off determination for all cells in a plotting area and apply both interval arithmetics and modular technique to the determinations. We implemented this method using CUDA programming environment. We investigated data structures and algorithms to be used for GPU processing and we describe a concrete method for efficient parallelization. The result of our experiment shows that the method is well suited for parallel computation, and further, compiling so as to keep *occupancy* value appropriate in order to hide memory access latency leads to speedup by 20%. Moreover, exponentiation computation of intervals turned out to be a bottleneck, which ought to be resolved, we believe, by mathematical improvement.

1. はじめに

陰関数グラフ描画は Risa/Asir[1] などの数式処理システムに実装されている機能の一つで，複雑な数式を視覚的に理解しやすいグラフとして表示させる，関数形の理解に役立つ機能である．陰関数の描画点（零点）を求める手法は複数存在するが，2) では描画空間を小さな矩形（以下，セルと呼ぶ）の集合とし，各セルの中に零点が存在するかを判定するという方法で，全ての描画点を求めグラフの形を得るという手法が提案されている．このとき，描画点を正確に求めるという観点から，整数演算によって零点を求める方法が示されている．また，各セルの評価を独立に行うことができることから，並列計算に適した問題であることが示されている．

近年，この問題のような並列性の高い計算を動作させるユニットとして，GPU（Graphics Processing Unit）が注目されており，GPGPU（General Purpose Computing on GPUs）と呼ばれる，GPU を様々な用途に使用する試みが盛んに行われている．GPU は画像処理を行う目的から単精度浮動小数点数演算に特化した設計となっており，GPGPU においても GPU の演算特性を活かした題材が頻繁に取り上げられている．このため，前述のような整数演算を主とする問題に対して使用される例は少ない．しかし，GPU の豊富な演算コアと高い並列性を活かすという観点では，必ずしも浮動小数点数演算である必要はないため，整数演算を主とする問題について GPU の使用を検討することも十分意義がある．

本研究ではこの手法を元に GPU に適したアルゴリズムを提案し，GPU 向け統合開発環境 CUDA[3] を用いて実装した．このプログラムにおける主要な計算は，各セルにおける関数値の計算である．各セルに割り当てられる領域は区間数[4] で表現され，対応する関数値も区間数として求められる．また，計算で使用する値は整数（有理数）として表現されるが，これに多倍長整数を扱う必要から各整数の保持にモジュラー表現[5] を用いている．実装の結果，この問題を GPU 上で高速に処理するためには，高い占有率を保つことが重要であることが分かった．一方，モジュラー表現化された数同士の比較処理や，区間数とモジュラー表現を組み合わせたことにより分岐処理が煩雑化したことが，処理のボトルネックとなっていることが確認された．

†1 電気通信大学電気通信学研究科情報工学専攻
Department of Computer Science, The University of Electro-Communications
†2 電気通信大学電気通信学部情報工学科
Department of Computer Science, The University of Electro-Communications

© 2010 Information Processing Society of Japan

以下，第 2 節では CUDA の概要について述べる．第 3 節では陰関数の描画点の求め方について述べる．第 4 節では GPU で動作させせることを考慮して実装したプログラムの詳細について述べる．第 5 節ではプログラムに適用した最適化について述べ，第 6 節で評価する．第 7 節では複数のカーネルに処理を分割する場合について述べる．第 8 節では本稿のまとめと今後の課題について述べる．

2. CUDA

本節では，CUDA におけるプログラミングの概要と，性能向上のための重要な点について述べる．プログラミングの詳細については 6) を参照されたい．

CUDA ではプログラムの制御構造としてスレッド，ブロック，グリッドが定義されている．スレッドは最小の制御単位で，GPU の各ストリーミングプロセッサ（SP）に割り当てられ実行される．ブロックはこのスレッドの集合であり，その処理はストリーミングマルチプロセッサ（SM）に割り当てられる．さらに，グリッドはこのブロックの集合であり，これが GPU に対する実行呼び出しの 1 単位である．GPU の 1 グリッドで実行するプログラムをカーネル関数と呼び，カーネル呼び出しを行う際にはカーネル関数をいくつのスレッド，ブロックで実行させるかを指定する．プログラムの実行はブロックごとに独立に行われるが，各ブロック内のスレッドはワープと呼ばれるスレッドの集合ごとに同一の命令コードによって動作する．このワープ内のスレッドが分岐によって別々の処理を行う場合，全てのスレッドによって双方のコードが実行されることになり，分岐条件に合わない処理の実行時にはスレッドの実行が待たされることになる．この現象はダイバージェント・ワープと呼ばれ，プログラムの性能低下につながる．

CUDA では GPU に存在する階層の異なる複数のメモリを使用することができ，それぞれグローバルメモリ，シェアードメモリ，コンスタントメモリなどのメモリモデルで扱うことができる．グローバルメモリは GPU と接続されたオフチップメモリの領域として保持され，全ブロック内のスレッドや CPU とデータを共有できる．グローバルメモリは容量が大きいが，オフチップであるためアクセス時のレイテンシが大きいという特徴がある．シェアードメモリは GPU の SM に搭載されているオンチップメモリであり，容量は小さいが高速なアクセスが可能で，同一ブロック内のスレッドで値を共有できる．コンスタントメモリはオフチップメモリに領域が確保されるが，GPU に搭載されているキャッシュによってデータがキャッシングされるため，グローバルメモリよりも高速にアクセスすることができる．また，コンスタントメモリには GPU からは書き込みができず，CPU からデータを割り当てる必要がある．

GPU の各 SM は複数のワープを同時に管理することができ，多数のワープを割り当てることでメモリアクセスのレイテンシを隠蔽することができる．このとき，同時にアクティブにできる最大のワープ数に対する，実際にアクティブになるワープ数の割合を占有率（Occupancy）と呼ぶ．占有率が低いと，SM に割り当てられるワープが減少し性能が低下する傾向にあるが，必ずしも占有率が高ければ性能が向上するとは限らない．しかし，どの程度計算資源を活用できているかの指標となるので，占有率を意識してプログラムを作成することが望ましい．

3. 陰関数描画とその算法

3.1 陰関数描画

陰関数 $f(x,y)=0$ のある範囲 $(x,y)=([x_{min},x_{max}],[y_{min},y_{max}])$ における零点を求め，計算機の表示装置にこの関数のグラフを描画することを考える．

零点を求める範囲（描画空間）に点は無数に存在し，描画空間上に零点が存在すればそれも無数に存在するが，表示装置にグラフを描画する場合，関数の零点は表示装置の最小単位（ピクセル）の各区間内に存在しているかどうかが分かれば十分である．そこで，描画空間は有限な大きさ $(x,y)=([x_{lo},x_{hi}],[y_{lo},y_{hi}])$ を持つセルで満たされていると仮定し，表示装置のピクセルをそれぞれセルに対応させ，セル中に零点が存在するかを判定する評価関数を適用し，その結果からグラフの描画点を得るものとする．すなわち，グラフの描画点を求めることは，零点が存在するセルの探索と見なすことができる．

零点の存在を判定する評価関数については様々な実装が考えられるが[4]，一般に陰関数は連続関数とは限らず孤立点が存在する可能性があり，正確に求めることは難しい．例えば，セルの四隅における関数値の符号から中間値の定理に基づき判定する方法が考えられるが，この場合セル内部に存在する孤立点を探索することができない．このため，本研究では区間数を利用した手法を使用して零点の判定を行う．これは，セルの範囲を x と y の区間とみなし，その区間における関数値を区間演算によって計算し，その値が 0 を含めば描画点とする方法である．これにより孤立点についても探索可能となり，正確なグラフを描くことが可能になる（**図 1**）．また，この評価は各セルごとに独立して行うことができるため，計算機で並列に実行することが可能である．

区間演算によって関数値を評価する際，扱うデータの型が問題になる．本研究では描画空間全体をセルが覆い尽くすことを想定している．しかし，浮動小数点数を使用する場合，

© 2010 Information Processing Society of Japan

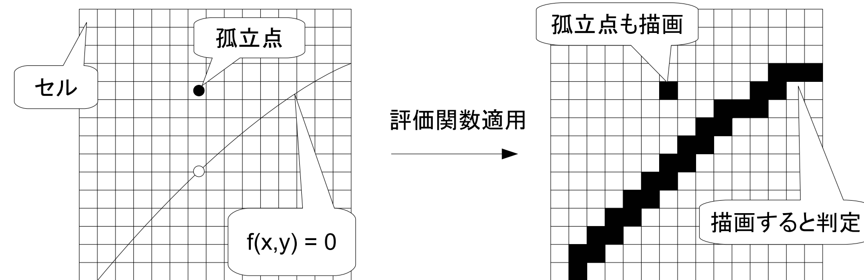

図 1 グラフの描画判定

境界の値を与えて計算する際に，丸めの方向などによって計算結果に違いが生じる可能性が考えられる．そのため，厳密にセルが連続しているように計算することが困難であり，正確に描画できない場合が発生する可能性がある．そこで，本研究では数を有理数で表現し，整数演算によって関数値を求める方法を採る[2]．この手法により，個々のセルの境界を正確に表現して描画点の判定を行うことができる．

有理数を用いて計算を行う場合に多倍長整数を扱うことになるが，通常の多倍長整数演算では扱うデータ領域のメモリ管理が必要となり，桁上がりの処理が逐次的になってしまうことからも並列処理には不向きである．この問題を解決するために，計算に使用する数は，中国人剰余定理に基づき複数の素数のもとでの剰余によって表現される「モジュラー表現」で表し，これを計算する「モジュラー算法」を用いて演算処理を行う方法を使用する[2]．

3.2 区 間 演 算

本研究で使用する区間数を用いた演算は,「区間数同士の加算，減算，乗算」,「区間数とスカラー数の乗算」,「区間数の冪乗算（指数は正整数）」である．区間数 $A = [a_1, a_2]$ と $B = [b_1, b_2]$ とスカラ数 c, n におけるこれらの演算を次のように定義する．

$$A + B = [a_1 + b_1, a_2 + b_2]$$
$$A - B = [a_1 - b_2, a_2 - b_1]$$
$$AB = [\min(a_1b_1, a_1b_2, a_2b_1, a_2b_2), \max(a_1b_1, a_1b_2, a_2b_1, a_2b_2)]$$
$$cA = [\min(ca_1, ca_2), \max(ca_1, ca_2)]$$

$$A^n = \begin{cases} [a_1^n, a_2^n] & (n \text{ が奇数}) \\ [a_2^n, a_1^n] & (a_1 \le a_2 < 0 \text{ かつ } n \text{ が偶数}) \\ [0, \max(a_1^n, a_2^n)] & (a_1 \le 0 \le a_2 \text{かつ } n \text{ が偶数}) \\ [a_1^n, a_2^n] & (0 < a_1 \le a_2 \text{かつ } n \text{ が偶数}) \end{cases}$$

このうち，加算と減算はスカラ数の演算 2 回で計算できるが，乗算と冪乗では場合分けが発生するため，複数の値を計算した上で合致するものを選ぶ必要がある．

3.3 モジュラー算法

3.3.1 モジュラー表現

整数 u を共通因子をもたないいくつかの法（通常は素数）$m_1, m_2, \cdots, m_r$ を用い，その剰余 $u_k = u \bmod m_k$ $(1 \le k \le r)$ を定め，数 u を $(u_1, u_2, \cdots, u_k)$ と剰余の列で表現する方法を「モジュラー表現」と呼ぶ[5]⋆1．また，モジュラー表現化された数をここでは「モジュラー数」と表記する．モジュラー表現化された整数 u は，$M = m_1m_2 \cdots m_r$ とすると，中国人剰余定理により $a \le u < a + M$ の範囲であれば一意に復元可能である．これより，モジュラー数化された整数 u は $a \le u < a + M$ の範囲であれば一意に復元可能である．a は表現したい値域によって変更することが可能で，元の整数とモジュラー数の間の変換の際に補正を行えばよい．本研究では，$a = -\lfloor M/2 \rfloor$ とし，$-\lfloor M/2 \rfloor \le u < \lfloor M/2 \rfloor$ を値域とする数について考える．

3.3.2 モジュラー数の復元

モジュラー数を元の整数に変換する算法は，Garner が 7) で示した手法が知られている．この算法では $1 \le i < j \le r$ に対する ${}_r\mathrm{C}_2$ 個の定数 m_{ij}^{-1} を用いる．ただし，

$$m_{ij}^{-1} m_i \equiv 1 \pmod{m_j}$$

である．この m_{ij}^{-1} は m_j を法とする m_i の乗法に対する逆元である（以降，この定数の集合を「法の逆数」と表現する）．m_{ij}^{-1} は拡張ユークリッドの互除法により計算が可能である．この法の逆数を用いて，r 個の整数 $v_1, v_2, \cdots, v_r$ を求める．算法を次に示す．

- $v_1 \leftarrow u_1$
- $k = 2, 3, \cdots, r$ の順に次を計算する
  - $v_k^{(1)} \leftarrow u_k$
  - $v_k^{(i+1)} \leftarrow (v_k^{(i)} - v_i) m_{ik}^{-1} \bmod m_k \quad (i = 1, 2, \cdots, k-1)$

⋆1 Residue Number System（RNS）とも呼ばれる．

© 2010 Information Processing Society of Japan

・ $v_k \leftarrow v_k^{(k)}$

この $(v_1, v_2, \cdots, v_r)$ の組を求めるアルゴリズムを，ここでは「混合基数変換」と表現する．このとき，$v_1, v_2, \cdots, v_r$ より，

$$u = v_1 + m_1(v_2 + m_2(\cdots(v_{r-1} + m_{r-1}v_r)\cdots))$$

となり，元の整数 u を求めることができ，これを「混合基数表現」と呼ぶ．また，$0 \le v_i < m_i$ ととれば $0 \le u < m$，$-\lfloor m_i/2 \rfloor \le v_i < \lfloor m_i/2 \rfloor$ ととれば $-\lfloor M/2 \rfloor \le u < \lfloor M/2 \rfloor$ の範囲で u を求めることができる．

3.3.3 モジュラー算法

モジュラー数における加算，減算，乗算を次に示す．

$c = a + b$ のとき，$c_k = a_k + b_k \bmod m_k \ (1 \le k \le r)$

$c = a - b$ のとき，$c_k = a_k - b_k \bmod m_k \ (1 \le k \le r)$

$c = a \times b$ のとき，$c_k = a_k \times b_k \bmod m_k \ (1 \le k \le r)$

これらの演算を「モジュラー算法」と呼ぶ．モジュラー算法は，各剰余ごとに独立に，簡単な計算によって行うことができる．

このモジュラー算法を区間演算に適用する場合，モジュラー数の符号判定と，2 数の大小判定が必要になる．これらはモジュラー数の剰余列の状態では行うことができないため，混合基数表現における $v_1, v_2, \cdots, v_r$ の列を $-\lfloor m_i/2 \rfloor \le v_i < \lfloor m_i/2 \rfloor$ の範囲で求めることで判定を行う[*1]．次に $(v_1, v_2, \cdots, v_r)$ で表される数の符号判定と，2 数 u_1 と u_2 の混合基数表現 $(v_{11}, v_{12}, \cdots, v_{1r})$ と $(v_{21}, v_{22}, \cdots, v_{2r})$ における大小比較法を示す．

- 符号判定：$v_r = v_{r-1} = \cdots = v_{k+1} = 0$ かつ $v_k \ne 0$ のとき，v_k の符号に等しい
- 大小比較：$k = r, r-1, \cdots, j+1$ について $v_{1k} = v_{2k}$ かつ
  - $v_{1j} > v_{2j}$ のとき，$u_1 > u_2$
  - $v_{1j} < v_{2j}$ のとき，$u_1 < u_2$

4. 実　　装

第 3 節で述べた算法やデータ表現法について整理し，実装したプログラムの詳細を解説する．

4.1 零点の探索法

零点を含むセルを求める方法として，2) ではまず複数のセルを含む領域について描画判定を行い，零点を含むと判定された領域に対してさらに小さな領域で描画判定を行っていくという手法を示している．グラフの描画点となるセルは，描画空間全体に存在するセルに対して極めて少ない場合が多いと予想されるため，計算量の削減という観点からは合理的である．しかし，この手法は分岐処理が頻発し，計算量も不均等であるため，GPU で行う処理としては不適当である．そこで，本研究では問題を単純化し，最初から全てのセルに対して個々に描画判定を行うという最も基本的な手法を採用した．また，各セルにおける関数値の計算はそれぞれ CUDA におけるスレッドで行い，1 スレッドの実行をもって 1 セルの描画判定が完結するものとした．

4.2 モジュラー数

GPU に搭載されているレジスタは 32 ビット長であるため，計算の際はその大きさを超えないことが望ましい．そのため，モジュラー数を構成する剰余列はそれぞれ符号なし 32 ビット整数（以降，uint 型整数と表記）とすることが妥当である．

剰余列 $(u_1, u_2, \cdots, u_r)$ にはそれぞれ対応する法 $m_1, m_2, \cdots, m_r$ が存在するが，法に 2^{16} 以上の数を取ると途中の演算で 32bit を超える数が出現するため，uint 型整数では表現できなくなる．これより，法には 2^{16} 未満の数を使用するものとした．法には素数を使用し，2^{16} 未満の素数の中から大きい順に r 個を法として使用した．これは，なるべく少ない法の個数で広い範囲の値域とするためである．

モジュラー数が使用する法は，モジュラー演算時に対応する値が参照できれば良いため，モジュラー数とは別に r 個の uint 型整数で保持しておけば良い．また，混合基数変換の演算時に使われる法の逆数 m_{ij}^{-1} については，あらかじめ各法から計算しておき，${}_r\mathrm{C}_2$ 個の uint 型整数として保持するものとした．

4.3 モジュラー算法

モジュラー算法は基本的に 3.3.3 節の定義と同様の実装で問題は無いが，各剰余を uint 型整数で保持しているため，減算の際に負の数にならないようにする必要がある．これを考慮して実際に実装した演算処理を次に示す．

- $c_k \leftarrow (a_k + b_k) \bmod m_k \ (1 \le k \le r)$
- $c_k \leftarrow ((a_k + m_k) - b_k) \bmod m_k \ (1 \le k \le r)$
- $c_k \leftarrow (a_k \times b_k) \bmod m_k \ (1 \le k \le r)$

4.4 セ　　ル

セルの大きさは x 方向，y 方向とも任意の大きさにすることが可能だが，描画装置に出力するという用途から，全て同じ大きさの正方形とすることが妥当である．1 辺の大きさ l は

*1 u そのものを求める必要はない．

© 2010 Information Processing Society of Japan

有理数で表現され，特に分子部分を l_{nume}，分母部分を l_{deno} として $l = l_{nume}/l_{deno}$ と表現できる．

セルはそれぞれ座標平面上の領域を持ち，その領域は $x = [x_{lo}, x_{hi}], y = [y_{lo}, y_{hi}]$ と区間数を用いて表現される．上限と下限の値は有理数で表現されるが，計算に使用する都合上，簡単のため分母部分は常に l_{deno} と考え，セルごとに与えられる領域のデータとしては $x_{nume} = [x_{lo} \times l_{deno}, x_{hi} \times l_{deno}], y_{nume} = [y_{lo} \times l_{deno}, y_{hi} \times l_{deno}]$ とし，分子部分のみを扱う．このとき，上限と下限の値はモジュラー数となるため，x_{nume}, y_{nume} は区間数の上限と下限をモジュラー表現化した数（以下,「区間モジュラー数」と呼ぶ）として保持される．

4.5 多項式の計算

n 個の項で構成される多項式 $f(x, y)$ は

$$f(x, y) = \sum_{k=1}^{n} c_k \times x^{e_{xk}} \times y^{e_{yk}} \quad (1)$$

と表され，係数 c_k と，x と y の指数 e_{xk}, e_{yk} によって具体的な形が決定される．

この多項式はモジュラー算法を用いた計算に使用されるため，係数 c_k はモジュラー数としてデータを保持し，x と y の指数 e_{xk}, e_{yk} は表現に多倍長整数が必要となる状況が稀であると考えられるため，通常の符号付き 32bit 整数型（以降，int 型整数と表記）で保持するものとした．また，c_k は一般に有理数で表現されるが，データ量の増大や計算の煩雑化を避けるため，あらかじめ各係数同士で通分された状態で保持するものとした．

式 (1) より，$f(x, y)$ の値は各項ごとに計算された値の和を求めれば良いことが分かる．この中で，変数 x と y は x_{nume}, y_{nume} として区間モジュラー数で与えられるため，$f(x, y)$ の値は区間モジュラー数で表現される．

セルに与えられた領域は分子部分である x_{nume}, y_{nume} のみが与えられるため，これらの冪乗を計算した後，分母 l_{deno} についても冪乗計算を行い通分する必要がある．このときの指数は，$f(x, y)$ の次数を o_f とすると，$o_f - (e_{xk} + e_{yk})$ となる．これより，実際の計算式は式 (1) を変形して次のようになる．

$$f(x, y) = \sum_{k=1}^{n} c_k \times x_{nume}^{e_{xk}} \times y_{nume}^{e_{yk}} \times l_{deno}^{o_f - (e_{xk} + e_{yk})} \quad (2)$$

ここで，第 k 項において式 (2) で計算される処理をまとめる．なお，T を計算結果を格納する変数，t_1, t_2, t_3 を一時変数とする（全て区間モジュラー数）．

(1) $t_1 \leftarrow x_{nume}^{e_{xk}}$（区間モジュラー数の冪乗）
(2) $t_2 \leftarrow y_{nume}^{e_{yk}}$（区間モジュラー数の冪乗）
(3) $T \leftarrow t_1 \times t_2$（区間モジュラー数と区間モジュラー数の乗算）
(4) $T \leftarrow T \times c_k$（区間モジュラー数とモジュラー数の乗算）
(5) $t_3 \leftarrow l_{deno}^{o_f - (e_{xk} + e_{yk})}$（モジュラー数の冪乗）
(6) $T \leftarrow T \times t_3$（区間モジュラー数とモジュラー数の乗算）

この T を n 個の項に対してそれぞれ求め，全て足し合わせることで得られる区間モジュラー数が，対応するセルの関数値である．

4.6 描 画 判 定

求められた関数値は区間モジュラー数で表されるため，区間中に 0 が含まれていれば，そのセル中に零点が存在すると判定することができる．従って，関数値を $[f_{lo}, f_{hi}]$ とすると，描画点であると判定される条件は,

$$sgn(f_{lo}) \times sgn(f_{hi}) \leq 0$$

となることから，区間数の上限と下限を示すモジュラー数に対して符号判定を行えば良い．

4.7 セルの割り当て

これまでに示した算法によって関数値の計算が行われるが，それぞれのスレッドは異なる x, y の区間を与えられることから，区間数の冪乗演算などで発生する分岐処理によってダイバージェント・ワープが発生する可能性がある．しかし，4.5 節で示した処理の中で，(1) と (2) の処理は与えられる変数の区間がワープ内で同じであれば行われる処理はワープ中の全スレッドで同一となり，ダイバージェント・ワープは発生しない．このため，本研究では同じブロック内のスレッドには同じ y の区間を持つセルを割り当てるものとし，(2) の処理におけるダイバージェント・ワープを回避するように配慮した．座標平面上のセルとスレッド，ブロック，グリッドとの対応を**図 2** に示す．

5. 最 適 化

第 4 節で実装したプログラムについて考えられる，アルゴリズムや GPU の特性を考慮した最適化手法について述べる．以下，最適化 1 と最適化 2 において，関数値の計算アルゴリズムにおける最適化，最適化 3 と最適化 4 において，GPU の特性を活かすための最適化を挙げる．また，具体的なパラメータについては，第 6 節で述べる．

5.1 混合基数変換の削減（最適化 1）

関数値を求める計算では主に区間モジュラー数の乗算や冪乗が行われるが，これらの演算

© 2010 Information Processing Society of Japan

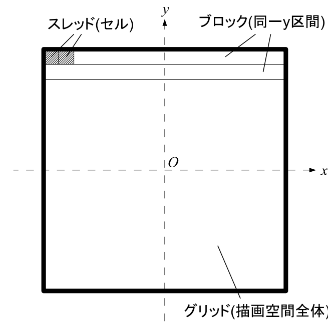

図 2 セルの割り当て

ではモジュラー数どうしの比較を行う必要がある．比較をするためにはモジュラー数に対して混合基数変換を行う必要があるが，この処理は $r \times (0+1+\cdots+(r-1))$ 回の 2 重ループとなっており，できる限り他の演算で置き換えることが望ましい．3.3.3 節で示したアルゴリズムは 2 数 u_1, u_2 の比較時において双方に混合基数変換を行うことを前提としているが，これは 2 数の差を求めて符号判定を行うという処理に置き換えることができる．すなわち，通常のモジュラー算法で $u' = u_1 - u_2$ を求め，u' の混合基数表現に対して符号判定を行い u_1 と u_2 大小関係を得るという方法である．これによって，混合基数変換の処理回数が 2 回から 1 回になるため，計算量が減り性能向上が期待できる．

5.2 モジュラー算法の除算命令削減（最適化 2）

モジュラー算法における加算，減算，乗算では除算（剰余演算）をする必要があるが，一般に除算はコストが高く，できる限り回避することが望ましい．そこで，これらの処理を次に示すように置き換えることを考える．

- $c_k \leftarrow a_k + b_k \ (1 \leq k \leq r)$
  - $c_k \geq m_k$ のとき，$c_k \leftarrow c_k - m_k$
- $c_k \leftarrow (a_k + m_k) - b_k \ (1 \leq k \leq r)$
  - $c_k \geq m_k$ のとき，$c_k \leftarrow c_k - m_k$
- $c_k \leftarrow a_k \times b_k \ (1 \leq k \leq r)$
  - $c_k \geq m_k$ のとき，$c_k \leftarrow c_k \bmod m_k$

これにより，分岐条件を満たさなければ除算に相当する計算をする必要が無くなるため，演算量の減少が期待できる．また，加算と減算では分岐する場合でも除算命令が減算命令となるため，より少ないコストで処理を行うことができる．

表 1 使用する陰関数

	Heart(x, y)	Spade(x, y)
次数	6	8
項数	16	25
係数の最大桁数（10 進）	10	124
必要な法の数	6	30
必要な法の逆数の数	15	435

5.3 レジスタ数の削減と占有率の増加（最適化 3）

第 2 節で述べた占有率は，1 ブロック当たりのスレッド数，1 スレッドが使用するレジスタ数，1 スレッドが使用するシェアードメモリの大きさによって決定される．このうち，使用するレジスタやシェアードメモリの量が減少すると，SM 内の資源が多くのワープに割り当てられるようになり，占有率は増加する．レジスタ数はプログラムのコンパイル時にオプションを与えることで上限を指定できるため，レジスタ数の上限を低く設定し占有率を高くすることによって，性能の向上が期待できる．

5.4 異なるメモリの使用（最適化 4）

第 4 節のプログラムでは，法 m_i，法の逆数 m_{ij}^{-1}，多項式の係数 c_k，多項式の変数 x, y の指数 e_{xk}, e_{yk}，セルの大きさの分母部分 l_{deno} の各データをグローバルメモリに配置し演算に使用している．オフチップであるグローバルメモリはアクセスコストが大きいため，演算の際にボトルネックとなっている可能性がある．このため，これらのデータを適切にシェアードメモリやコンスタントメモリに配置することで，アクセスコストが減少し性能の向上が期待できる．

6. 評価実験

第 5 節で述べた最適化について評価実験を行った．実験には GT200 コアを搭載する GPU である NVIDIA GeForce GTX 285 を使用し，プログラムの作成には CUDA 2.30 を使用した．陰関数には，Risa/Asir の標準ライブラリに実装されているハート関数 Heart(x, y) とスペード関数 Spade(x, y) を使用した[⋆1]．これらの関数のパラメータを**表 1** に示す．

実験では描画空間を $[x_{min}, x_{max}] = [y_{min}, y_{max}] = [-128/100, 128/100]$，セルの大きさを $l = 1/100$ と設定した．また，これに伴い 1 グリッドあたりのブロック数は 256，1 ブ

⋆1 それぞれ，トランプのハートとスペードの形をしたグラフとなる．

© 2010 Information Processing Society of Japan

表 2 プログラムの実行時間 (msec)

最適化 1	最適化 2	最適化 3	最適化 4	Heart	Spade
				285.7	5420.8
○				237.3	2809.9
○	○			231.8	2749.3
○	○	○		224.6	2298.2
○	○	○	○	196.6	2206.4

ロックあたりのスレッド数は 256 とした．プログラムには最適化 1 から最適化 4 まで順に適用し，最も性能向上が見られた実装を継承しながら評価を行った．この実験の結果を**表 2**に示す．以下，適用した最適化の詳細について述べる．

最適化 1 では，ハート関数において 20%，スペード関数において 93% の速度向上を確認した（表 2）．$O(r^2)$ の計算量となる処理を削減したことで，より法の数 r が大きいスペード関数において効果が顕著に現れたが，同時に関数値の計算においてこの処理の占める割合がとても大きいということも分かる．

最適化 2 では，加算と減算に適用した場合に速度向上が確認できたが，乗算の場合は逆に実行速度が低下する結果となった．乗算ではワープ内で全スレッドが分岐条件を満たさない場合が少ないため，分岐コストの増加のみ現れてしまったと考えられる．また，加算と減算のみを適用した場合には，ハート関数，スペード関数ともに約 2% の速度向上となった（表 2）．この最適化では演算コストが小さく，分岐コストも増加することから，最適化 1 と比べて小さな効果にとどまったと推定される．

最適化 3 では，最適化 2 までを適用したプログラムのレジスタ数が 21（占有率 0.5）であったことから，最大レジスタ数を 20（同 0.75），16（同 1）と設定した場合について実験を行った．この結果，占有率が 0.75 の場合において，ハート関数で 3%，スペード関数で 20% と最も大きい速度向上を確認した（表 2）．占有率が 1 の場合についても速度向上は見られたが，0.75 の場合よりもその効果は小さくなった．これはレジスタ数を減らしたことによって，本来レジスタに保持される値がオフチップメモリに待避されるようになるため，アクセス時のレイテンシが大きくなった結果と考えられる．このため，プログラムによって最適な占有率を設定すること重要であることが分かる．

最適化 4 では，各データをシェアードメモリとコンスタントメモリに割り当てた場合について実験を行った結果，法 m_i，法の逆数 m_{ij}^{-1} ではどちらに割り当てた場合でも速度向上が見られ，特に m_i はコンスタントメモリ，m_{ij}^{-1} はシェアードメモリに置いた場合に最も実行時間が短くなった．しかし，その他のデータの場合は速度向上効果は現れなかった．効果の無かったデータは計算中に参照する回数が少なく，アクセスコストの低下が非常に小さかったことが原因と考えられる．この結果から，m_i をコンスタントメモリ，m_{ij}^{-1} をシェアードメモリに割り当てる場合が最適と考えられるが，m_{ij}^{-1} は法の数 r が大きくなるとデータ量が 2 次関数的に増加するため，シェアードメモリの割り当て量増加に伴う占有率の低下が発生する場合がある．占有率の低下は $r \leq 52$ の場合に発生し，この場合実行速度が大幅に低下してしまう．また，m_{ij}^{-1} はコンスタントメモリに割り当てた場合にも速度向上効果が見られ，シェアードメモリの場合との差も少ない．このことから，m_i と m_{ij}^{-1} のデータは，双方ともコンスタントメモリに割り当てる場合が適当と判断し実装を行った結果，ハート関数で 14%，スペード関数で 4% の速度向上を得た（表 2）．

7. 複数カーネルによる計算

GPU を使用して計算を行う場合，ドライバの制限に起因し，実行時間が長くなるとカーネル関数の実行が途中で打ち切られてしまう場合がある．このため，1 回のカーネル実行に時間がかかる描画空間の大きな問題については，計算を複数のグリッドに分割する必要がある．この場合，できる限り分割数を少なく抑え，なおかつ各カーネルの占有率を高く保つことが重要であるが，描画空間として同じ面積の範囲を割り当てているにもかかわらず計算時間が変化する場合がある．この例として，描画空間が $[x_{min}, x_{max}] = [y_{min}, y_{max}] = [-384/100, 384/100]$，セルの大きさが $l = 1/100$ である場合について，スペード関数の描画点を求める場合を挙げる．この描画範囲を 1 グリッド当たりのブロック数が 768，1 ブロックあたりのスレッド数が 256 となるように分割し，それぞれの計算範囲として**図 3** に示す範囲が与えられた場合の実行時間を**表 3** に示す.

表 3 各カーネルの実行時間 (msec)

カーネル	x_{nume}	y_{nume}	実行時間
カーネル 1	$[-384, 384]$	$[192, 384]$	6004.8
カーネル 2	$[-384, 384]$	$[-192, 192]$	6050.2
カーネル 3	$[-384, 384]$	$[-384, -192]$	6013.2
カーネル 4	$[192, 384]$	$[-384, 384]$	4117.2
カーネル 5	$[-192, 192]$	$[-384, 384]$	6004.5
カーネル 6	$[-384, -192]$	$[-384, 384]$	4125.8

この結果から，カーネル 4 とカーネル 6 のみが，その他のカーネルと比べて実行時間が 2/3 程度であることが分かる．この原因は,（区間）モジュラー数の冪乗演算においてダイ

© 2010 Information Processing Society of Japan

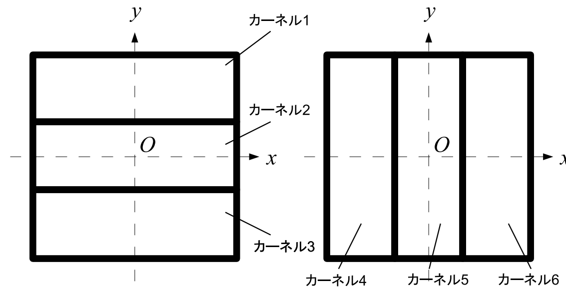

図 3 各カーネルの計算範囲

バージェント・ワープが頻発していることである．特に区間数の冪乗計算は分岐判定が煩雑であり，x としてどのような区間が与えられてもダイバージェント・ワープが発生するが，x が 0 に近い値ではこれが顕著に現れるために実行時間が増加している．このため，現状では描画空間の分割は y 軸方向に対して行い，できる限り $x = 0$ をまたぐ区間を少数のカーネルに割り当てることが，最も性能を高く保つことができる分割手法となる．

8. まとめと今後の課題

本稿では，区間数と整数演算を利用した陰関数のグラフ描画点判定問題の実装を GPU に対して行い，実装する上でのデータ構造や算法に関する指針を示した．また，第 6 節の評価実験の結果から，本研究の実装では占有率が演算性能に大きな影響を与えることが確認した．本実装では，計算の際に必要な一時変数がモジュラー数となる場合が多く，複数の整数で構成されるデータをレジスタやシェアードメモリでは保持することができない．このため，一時変数はオフチップメモリに保持されることとなり，レイテンシの高いメモリへのアクセスが頻発することになる．このような特性により，高い占有率を保つことによるレイテンシ隠蔽効果が強く作用し，結果的に性能向上に結びついたと考えられる．

今後の課題としては，アルゴリズムの改善が挙げられる．第 6 節では混合基数変換の回数を減らすことで速度を向上させることができたが，計算量が $O(r^2)$ であることから，特に法数 r が大きい場合について計算上のボトルネックとなる．このため，この処理を行う目的であるモジュラー数の比較を少ない計算量で行うことができれば，更なる性能向上が期待できる．また，第 7 節で示した（区間）モジュラー数の冪乗計算も改善の余地がある．$x = 0$ 付近の値による冪乗計算はグラフの平行移動によって回避できるが，平行移動をすると項の数が増加してしまうため，全体の処理性能が低下してしまう可能性があるため，可能であれば冪乗計算のアルゴリズム改善によってダイバージェント・ワープを回避することが望ましい．

計算に使用する区間演算についても改善の余地がある．本研究で使用しているアルゴリズムは必要最低限の単純なものであるため，計算によって区間の膨張が発生し，本来零点を含まないセルについても零点と判定してしまう．このため，本来は曲線でグラフが表現される箇所が面として現れる場合があるため，今後は数理的な観点からもアルゴリズムの改良をする必要がある．

参 考 文 献

1) 神戸大学 : Risa/Asir (神戸版). `http://www.math.kobe-u.ac.jp/Asir/asir-ja.html`.
2) 近藤祐史，村尾裕一，斎藤友克 : Risa/Asir の ifplot の改良と並列化の試み，京都大学数理解析研究所講究録 1568，pp.185–191 (2007).
3) NVIDIA Corporation: CUDA Zone, `http://www.nvidia.com/object/cuda_home.html`.
4) 齋藤友克，　竹島卓，平野照比古 : グレブナー基底の計算 実践篇 Risa/Asir で解く，東京大学出版会 (2003).
5) Knuth, D.E.: *The Art of Computer Programming*, Vol.2, Addison Wesley, third edition (1998).
6) NVIDIA Corporation: NVIDIA CUDA Programming Guide Version 2.3.1 (2009).
7) Garner, H.L.: The residue number system, *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference*, pp. 146–153 (1959).

© 2010 Information Processing Society of Japan