GPU による高速画像処理

名古屋大学大学院情報科学研究科

出口 大輔，井手 一郎，村瀬 洋

概要： 本発表では，近年注目を集めている GPGPU（General Purpose computing on GPUs）の技術に着目し，GPGPU を利用するための開発環境の使い方やプログラミングのノウハウを分かりやすく解説する．GPGPU は GPU を汎用計算に利用しようという試みであり，現在では物理シミュレーション，数値計算，信号解析，画像処理・認識などの分野で広く利用されるようになっている．ここ数年では，GPGPU を行うための開発環境の整備も進んできている．そこで本発表では，NVIDIA 社が提供している開発環境の CUDA を題材に取り上げ，簡単に GPGPU を行うことができることを示すと供に，計算コストの高い画像処理が GPU を利用することで大幅に高速化できることを示す．

1. はじめに

近年の GPU の高性能化に伴い，GPU を汎用計算に利用しようという試みである GPGPU が注目を集めている．図 1 は 2003 年～2008 年に発売された CPU と GPU の性能差を GFLOP/s（1 秒あたりの浮動小数点演算性能）で表したものである．図から分かるように，2003 年頃は CPU と GPU の差はそれほど大きくはないが，2008 年にはその差が約 10 倍程度に広がっている．また，GPU は非常に高性能であるにもかかわらず，GeForce GTX 285（現在発売されている GPU の中で最も高性能なものの一つ）は 4 万円～5 万円程度で購入することができる．このように，非常に高性能な GPU を安価に入手できるようになった点も，GPGPU の技術が注目される大きな要因となっている．

このように，CPU と比較して非常に高性能な GPU をグラフィックス以外の処理へ利用しようという試みは，2003 年にプログラマブルシェーダが登場して以降，広く行われるようになった[1][2]．これは，プログラマブルシェーダを利用することにより GPU 上でプログラムを比較的簡単に動作させることが可能になったためである．しかしながら，プログラマブルシェーダを利用する

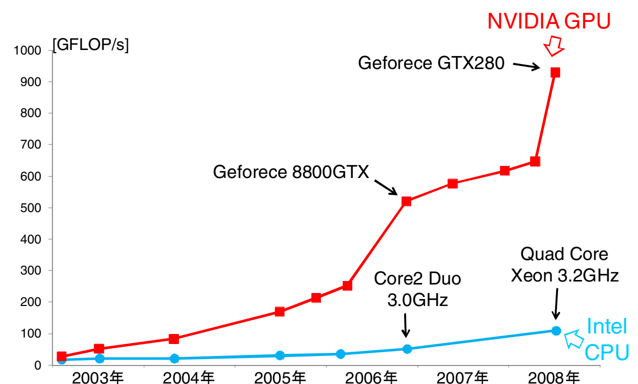

図 1　CPU と GPU の性能比較[3]

ためにはグラフィックスパイプラインを強く意識したプログラミングが必要なため，GPGPU の敷居は必ずしも低くなかった．この問題を解決するため，GPGPU を行うための開発環境の整備が進んでおり，NVIDIA 社の CUDA，AMD 社の ATI Stream などが利用できるようになっている．これらの開発環境では，グラフィックスパイプラインを意識することなく GPU が利用できるため，非常に簡単に GPGPU を行うことが可能である．

そこで本稿では，開発環境として NVIDIA 社の CUDA を取り上げ，CUDA でプログラミングを行う際に必要となる環境構築方法，およびプログラミング時の注意点を分かりやすく解説する．また，計算コストの高い画像処理が GPU を利用することで簡単に高速化できることを示す．

2. CUDAの環境構築

本節では，まず CUDA の概要を説明した後，CUDA を実行するために必要となる環境（OS,グラフィックスカードなど），CUDA を実行するために必要となるソフトウェアの導入方法を順に説明する．

2.1 CUDAとは？

NVIDIA 社が提供する CUDA は Compute Unified Device Architecture の略であり，C/C++言語を利用して GPU の処理を記述することができる統合開発環境である．従来，GPGPU を行うためには, HLSL や GLSL といったシェーダ言語を利用する必要があり，グラフィックスパイプラインを強く意識したプログラミングが必要であった．そのため，グラフィックス処理に合わせたアルゴリズムの変更が必要不可欠であり，これが GPGPU の敷居を高くしていた大きな原因であった．これに対し，CUDA では通常の C/C++言語の関数を呼び出す（スレッドで処理を行う）感覚で GPU を利用できるようになっている．そのため，グラフィックスパイプラインに関する知識は不要であり，C/C++言語を学習したことのある研究者や開発者であれば比較的容易に GPGPU を行うことが可能になっている．また，C/C++言語を用いて GPGPU を行うことが可能であるため，既存のアルゴリズムの移植も容易であるという特徴がある．以下では，CUDA を利用するための準備段階として，環境の構築方法を説明する．

2.2 実行環境の準備

CUDA を利用するためには，表 1 に示す NVIDIA 社製のグラフィックスカードを用意する必要がある．例えば，GeForce GTX 285 はコンシューマー向けのグラフィックスカードの中ではかなり高性能なものであり，その性能は約 1 TFLOP/s である．GeForce GTX 285 は非常に高性能ではあるが，その市場価格は 4 万円～5 万円程度であるため，比較的容易に入手できるのではないだろうか．また，Quadro シリーズはワークステーション等で利用されている非常に高価なグラフィックスカードであるが， GeForce シリーズと大きな性能差は無い（ただし，搭載されているメモリ量は Quadro シリーズの方が多い）．そして, Tesla は CUDA を実行するための専用ハードウェアであり，ビデオ出力は搭載されていない．初めて GPGPU に挑戦するのであれば，安価な GeForce シリーズがお勧めである．

表 1　CUDA 対応のグラフィックスカード

GeForce	GTX 280, GTX 260, 9800 シリーズ, 8800 シリーズ，他
Quadro	Plex 2200 D2, FX 5800, FX 5600，他
Tesla	S1070, C1060, S870, D870, C870

CUDA が対応している OS は，Windows XP, Windows Vista , Windows Server 2008 , Windows 7，Linux，Mac OS であり，現在広く普及しているさまざまな OS 上で実行することが可能である．本稿では説明の都合上，Windows をメインの実行環境として説明を行う．Linux や Mac を使われている方々は，CUDA Zone [4] を参考に環境構築を行っていただきたい．

2.3 開発環境の準備

CUDA を利用して GPU のプログラムを作成するためには，2.1 で説明したハードウェアの準備に加え，本節で述べる開発環境の準備が必要となる．まず，CUDA Zone [4] へアクセスし，CUDA Driver，CUDA Toolkit，CUDA SDK をダウンロードしよう．CUDA Driver は CUDA 対応のビデオドライバーであり, CUDA 上でプログラムを動作させるために必要なものである. CUDA で作成したプログラムを配布する際には，配布先でも CUDA 対応のビデオドライバーが必要になるので注意が必要である. CUDA Toolkit は CUDA の開発で利用する nvcc コンパイラや CUBLAS や CUFFT といった数値計算ライブラリ，プログラミングガイド等のドキュメントが含まれている．

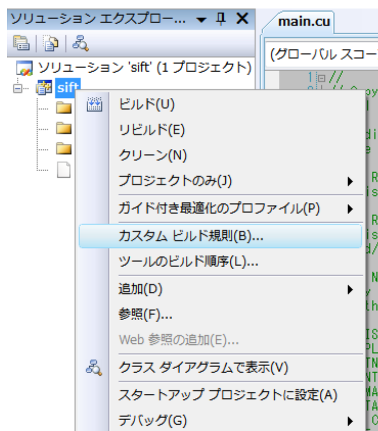

図 2 「カスタムビルド規則」の選択

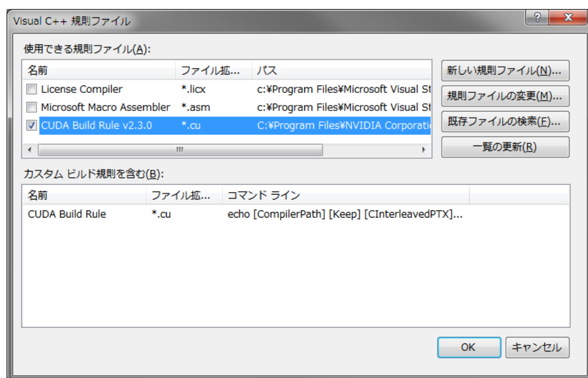

図 3 「Cuda.Rules」の追加

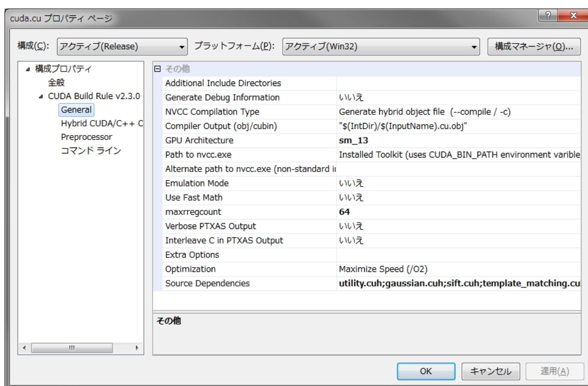

図 4 NVCC のコンパイルオプション

CUDA で開発を行う際は必ず必要になるため，ドライバーと併せてインストールして欲しい．最後に，CUDA SDK には CUDA を利用する上で非常に参考になる多くのサンプルプログラムがソース付きで収録されている．初めて CUDA に触れる際は，一通り目を通すことを強くお勧めする．また，Visual Studio を使って CUDA のプログラムを開発する際に有用なツールである「Cuda.Rules」も含まれている．そのため，CUDA Driver と Toolkit に加えて SDK もインストールしておくことをお勧めする．

本稿の執筆時点（2009 年 8 月）では，CUDA のバージョンは 2.3 である．以降は，バージョン 2.3 を利用して話を進める．

2.4 Visual Studio の設定

Visual Studio は Microsoft 社が販売している統合開発環境であり，世界中で多くの研究者・開発者が利用しているツールである．そこで，本節では CUDA の開発を Visual Studio で行う際の設定方法を説明する．まず本節を読み進める前に，2.3 で述べた CUDA SDK のインストールを済ませて欲しい．

Visual Studio の新規作成画面から Visual C++ の Win32 コンソールアプリケーションもしくは Win32 プロジェクトを作成する．そして，図 2 に示すプロジェクトメニューから「カスタムビルド規則」を選択し，図 3 の「既存ファイルの検索」を実行する．その際，CUDA SDK に含まれる「Cuda.Rules」を選択しよう．これらの操作により，図 4 に示すようなプログラムをコンパイルする際のオプションを Visual Studio 上で設定できるようになる．ただし，これらのオプションはファイルの拡張子が “.cu” のものに対してのみ有効であるので注意が必要である．プロジェクトをビルドする際は，追加のライブラリパスに “$(CUDA_LIB_PATH)” を設定し，cuda.lib と cudart.lib の 2 つを追加の依存ファイルに設定する．これにより，CUDA が利用可能になる．

3. CUDA のプログラミングモデル

CUDA は，GPU を多数のスレッドが高い並列性を持って処理を実行できるデバイスとして扱う． GPU は数百ものスレッドを並列に処理することが可能なため，CUDA にはスレッドを階層的に管理する仕組みが導入されている．また，GPU 上に実装された特殊なメモリ領域へのアクセス

方法や，スレッド間の同期，といった機能も簡単に利用できるようになっている．以下では CUDA でプログラムを書く前段階として，CUDA のスレッド管理の仕組み，CUDA のメモリモデルについて説明する．

3.1 スレッド管理の仕組み

CUDA では，図 5 に示すようにブロックとグリッドの 2 つの階層でスレッドを管理する．具体的には，スレッドのまとまりをブロックと呼び，ブロックのまとまりをグリッドと呼ぶ．また，スレッドおよびブロックは 3 次元的に配置することが可能であり，各スレッドおよび各ブロックは X，Y，Z の 3 つの整数の組で一意に識別される．つまり，各スレッドおよび各ブロックの ID は (1, 0, 0) や (1, 2, 3) のように表される．そして，CUDA ではグリッドを一つの単位として処理を実行する．具体的には，図 6 に示すように処理毎にスレッドとブロックの数を決定して処理を実行する．また，GPU の処理結果を CPU 側で利用する場合は，GPU と CPU の同期をとった後でメモリ転送が必要となる．

これらのスレッド管理の機構は GPU の構成と密接な関係があり，同じブロック内のスレッド同士でしか処理の同期ができない（ブロック間でスレッドの同期をとる場合は CPU の処理が必要）といった制限が存在する．これらの詳細に関しては文献[3]を参照されたい．

3.2 メモリモデル

GPU には CPU でプログラムを書く場合には利用しない，特殊なメモリ領域がいくつか存在する．具体的に GPU が利用できるメモリは，レジスタ，ローカルメモリ，共有メモリ，コンスタントメモリ，テクスチャメモリ，グローバルメモリ，の 6 種類である（図 7）．これらのメモリは大きく分けて，(1) スレッド内でのみ利用可能なメモリ，(2) ブロック内のスレッドで共有されるメモリ，(3) すべてのスレッドで共有されるメモリ，の 3 つに分けられる．レジスタとローカルメモリは (1) に対応し，共有メモリが (2) に対応する．それ

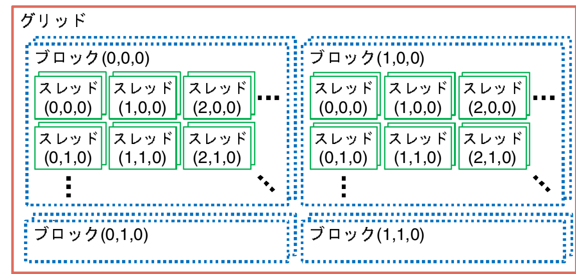

図 5　階層的なスレッド管理

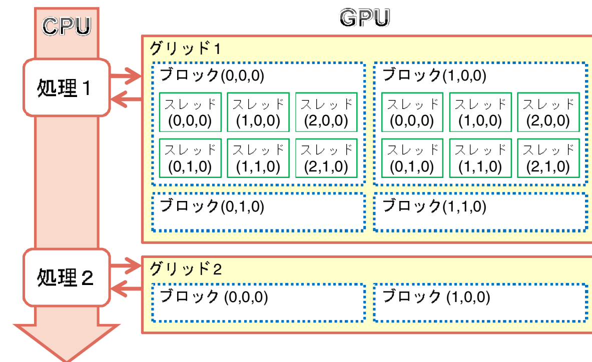

図 6　CUDA における計算の流れ

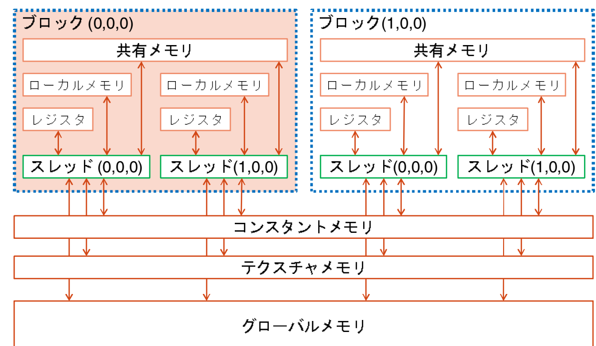

図 7　CUDA のメモリモデル

以外は (3) となる．使用するメモリによってはスレッド間で共有できないものもあるため注意が必要である．また各メモリ領域は，メモリ量，アクセス速度，キャッシュの有無，などに違いが存在するため，目的に応じて使用するメモリを適切に選択する必要がある．例えば，グローバルメモリは大容量であるが，1 回のメモリアクセスに 400〜600 クロックサイクルが必要である．そのため，Coalesced メモリアクセスの考慮が必要となる．これらの詳細は文献[3]を参照されたい．

CUDA で利用可能な 6 種類のメモリの内，テクスチャメモリは GPU に特有のメモリ領域である．

```
 1:  #include <stdio.h>
 2:
 3:  __global__ void hello( char *data )
 4:  {
 5:      char *text = "Hello World!!¥n";
 6:      data[ threadIdx.x ] = text[ threadIdx.x ];
 7:  }
 8:
 9:  int main( int argc, char *argv[] )
10:  {
11:      char *dData, hData[ 14 ];
12:      cudaMalloc( ( void ** )&dData, sizeof( char ) * 5 );
13:
14:      dim3 nThreads( 14, 1 );
15:      dim3 nBlocks( 1, 1 );
16:      hello<<< nBlocks, nThreads >>>( dData );
17:
18:      cudaMemcpy( hData, dData, 14,
                                  cudaMemcpyDeviceToHost );
19:
20:      for( int i = 0 ; i < 14 ; i++ )
21:      {
22:          printf( "%c", hData[ i ] );
23:      }
24:
25:      cudaFree( dData );
26:      return( 0 );
27:  }
```

図 8　CUDA で“Hello World!!”

テクスチャメモリでは，2 次元テクスチャに対して効率の良いキャッシュ機構や，ハードウェア線形補間，といった CPU に無い機能を利用することができる．画像処理のアルゴリズムを高速化する上で，テクスチャメモリは非常に使いやすく重宝するメモリ領域である．具体的な使い方は以降の節を参照していただきたい．

4. CUDA で“Hello World!!”

本節では，“Hello World!!”をサンプルとして利用し，CUDA における基本的なプログラミング方法を紹介する．図 8 は CUDA を利用して GPU 上で“Hello World!!”を計算するプログラム例である．本プログラムをテキストエディタに入力し，“hello.cu”という名前で保存しよう（拡張子“.cu”は GPU 上で実行されるコードを含むファイルを

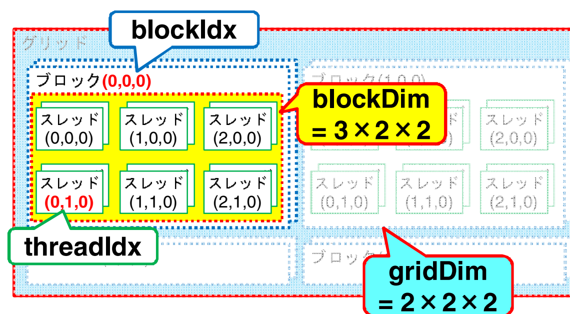

図 9 スレッド数とブロック数の取得方法

表す拡張子である）．そして「Visual Studio 2008 コマンドプロンプト」を立ち上げ，「nvcc hello.cu」と打ち込むと“Hello World!!”を表示するプログラムが生成される．もちろん，2.4 で設定した Visual Studio のプロジェクトを用いても“Hello World!!”をコンパイルすることができる．生成されたプログラムを実行すると，コンソールに“Hello World!!”と表示されるはずである．一見，通常の C/C++のプログラムと変わらないように思われるかもしれないが，本プログラムには CUDA で GPU を利用する際に必要となる重要な点がいくつか含まれている．以下でそれらの具体的な説明を行う．

図 8 をよく見ると，“__global__”や“threadIdx”といった通常の C/C++言語では見慣れない記号がいくつか含まれていることが分かる．これらは，CUDA で GPU を利用するために行った C/C++言語の拡張部分である．まず，“__global__”は CPU から呼び出され GPU 上で実行される関数を表す修飾子である．“__device__”や“__host__”なども存在するが，それらの詳細は文献[3]に譲る．そして，“__global__”が付与された関数を CPU から呼び出す際は，図 8 の 16 行目のように関数名と引数の間に“<<< >>>”で囲まれたパラメータを指定する．これは，3.1 で説明したスレッドとブロックの配置方法を指定するものである．具体的には，14～15 行目で設定したスレッド数とブロック数により GPU 上で生成されるスレッド配置が決定される．このように，GPU で実行される関数を CPU から呼び出す際は，生成するスレッド数を指定して関数を呼び

出す必要がある点に注意が必要である．GPU 上の各スレッドは，6 行目の “threadIdx” という組み込み変数を参照することで，自身の ID（X, Y, Z の 3 つの整数の組）を把握することができる．“threadIdx” 以外にも，図 9 に示すような変数を用いることでブロックの ID（blockIdx），ブロック内のスレッド数（blockDim），グリッド内のブロック数（gridDim）を取得することができる．これらの変数は各スレッドの計算範囲を決定するために利用することができる．

最後に，GPU からは CPU 上のメモリ（主記憶）にアクセスできないという制限がある．そのため，GPU 上で利用するメモリ領域は別途確保する必要がある．これを行っているのが図 8 の 12 行目である．また，CPU と GPU で処理するデータを共有する場合は，18 行目の “cudaMemcpy” 関数を利用して互いにメモリ転送を行う必要がある．ただし，使用するメモリ領域によっては異なる関数を使用する必要があるため，文献[3]をよく読んでから利用して欲しい．本操作は CPU でプログラムを書く際には意識する必要のないものである．そのため，GPU で処理を実行する場合はどのメモリ領域を利用しているかに関して十分な注意が必要である．

5. テンプレートマッチングの実装

テンプレートマッチングは画像処理の分野で広く用いられている基本的な手法であり，基盤の品質検査や画像中の特定物体（人物など）の検出に利用されている．テンプレートマッチングは，入力画像中に窓を設定し，その窓の大きさと位置を変化させながらテンプレートとの類似度を評価することで最もテンプレートに類似する部分を見つける処理である．しかしながら，窓の位置と大きさをさまざまに変化させながら類似度評価を行うため，膨大な数の類似度評価が必要となる．そのため，一般的にテンプレートマッチングをリアルタイムで動かすことは難しく，これまでにさまざまな高速化手法[5]が提案されている．本節では，実時間で動作させることの難しかったテンプレートマッチングを GPU 上に実装し，リアルタイムに動作させる方法を説明する．

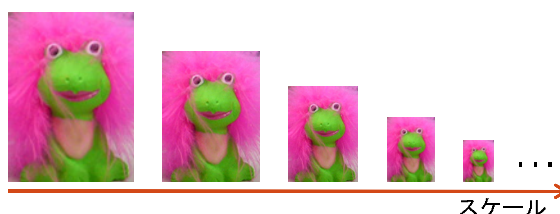

図 10　複数スケールのテンプレート

5.1 高速化の基本戦略

テンプレートマッチングでは，類似度の評価は窓単位で独立に行うことが可能である．これは，膨大な数の処理を同時に実行できる GPU に適した問題である．そこで，各窓の類似度評価を GPU 上の各スレッドが行うことにより，類似度評価の並列化を図る．

テンプレートマッチングには，隣り合う窓同士はほぼ同じ部分画像（メモリ領域）にアクセスするという特徴がある．そのため，3.2 で説明した大容量のグローバルメモリを利用する場合は，メモリアクセスが大きなボトルネックとなる．そこで，本節では GPU 特有の機能であるテクスチャメモリを用いることでメモリアクセスの効率化を図る．テクスチャメモリは読み取り専用のメモリ領域ではあるが，2 次元画像に対して効率的にキャッシュが可能な機構を備えている．このキャッシュ機構を利用することによりメモリアクセスの高速化を図る．また，テンプレートマッチングでは窓の大きさを変化させながら類似度評価を行うため，図 10 のように複数スケールのテンプレートを事前に用意する必要がある．本節では，複数スケールのテンプレートを事前に用意する代わりに，テクスチャメモリの正規化座標を利用することでスケールの変化に対応する．以下では，具体的なテンプレートマッチングの実装方法を示す．

5.2 実装方法

まず，テンプレートマッチングで利用するテンプレートおよび入力画像をテクスチャメモリに

```
1:  texture< uchar4, 2, cudaReadModeElementType > imgTex;
2:  texture< uchar4, 2, cudaReadModeNormalizedFloat > refTex;
3:
4:  __global__ void kernel( /* 引数は省略 */ )
5:  {
6:     int i = threadIdx.x + blockDim.x * blockIdx.x;
7:     int j = threadIdx.y + blockDim.y * blockIdx.y;
8:
9:     float err = 0.0f;
10:
11:    if( i < areaW && j < areaH )
12:    {
13:       float _1_w = 1.0f / maskW;
14:       float _1_h = 1.0f / maskH;
15:       for( int n = 0 ; n < maskH ; n++ )
16:       {
17:          for( int m = 0 ; m < maskW ; m++ )
18:          {
19:             uchar4 p1 = tex2D( imgTex, i + m, j + n );
20:             float4 p2 = tex2D( refTex, _1_w * m, _1_h * n ) * 255.0f;
21:             err += ( p1.x - p2.x ) * ( p1.x - p2.x );
22:             err += ( p1.y - p2.y ) * ( p1.y - p2.y );
23:             err += ( p1.z - p2.z ) * ( p1.z - p2.z );
24:          }
25:       }
26:
27:       err *= _1_w * _1_h;
28:       if( error[ i + j * imgW ] > err )
29:       {
30:          error[ i + j * imgW ] = err;
31:          scale[ i + j * imgW ] = s;
32:       }
33:    }
34: }
35:
36: // GPU 側に入力画像用のメモリ領域を確保する
37: cudaArray *iArray;
38: cudaChannelFormatDesc c1
                        = cudaCreateChannelDesc< uchar4 >();
39: cudaMallocArray( &iArray, &c1, 画像の幅, 画像の高さ );
40: cudaMemcpyToArray( iArray, 0, 0, 入力画像のポインタ,
           入力画像のバイト数, cudaMemcpyHostToDevice );
41: cudaBindTextureToArray( imgTex, iArray, c1 );
42:
43: // GPU 側にテンプレート画像用のメモリ領域を確保する
44: cudaArray *rArray;
45: cudaChannelFormatDesc c2
                        = cudaCreateChannelDesc< uchar4 >();
46: cudaMallocArray( &rArray, &c2, 画像の幅, 画像の高さ );
47: cudaMemcpyToArray( rArray, 0, 0, テンプレートのポインタ,
           テンプレートのバイト数, cudaMemcpyHostToDevice );
48: cudaBindTextureToArray( refTex, rArray, c2 );
49:
50: // 正規化座標を有効にする
51: refTex.filterMode = cudaFilterModeLinear;
52: refTex.normalized = 1;
```

図 11　テンプレートマッチングの主要部

確保するための準備を行う．　テクスチャメモリを利用するためには，

texture<Type, Dim, ReadMode> 変数名;

によりテクスチャメモリを表す変数を一つ用意する必要がある．ここで，Type はテクスチャ内の各画素の型であり int，float，int3，float4 等が指定可能である．また，Dim はテクスチャの次元を表し，1～3 のいずれかを指定する．そして，ReadMode はテクスチャメモリからの読み出し時に値を正規化するかどうかを示すフラグである．cudaReadModeElementType を指定した場合は各データ型に対応した値が返され，cudaReadModeNormalizedFloat を指定した場合は値が 0～1 の範囲に正規化される（Type が符号付きの場合は－1～1）．今回は 2 次元画像が対象であるため，Type に uchar4，Dim に 2 を指定している（図 11 の 1～2 行目）．

図 11 の 36～48 行目が GPU 上のメモリ領域の確保とテクスチャメモリへのマッピングを行う部分である．そして，50～52 行目により正規化座標を有効にしている．ここで，図 12 は正規化座標を有効にした場合としない場合の違いを示している．図 12 と図 13 から分かるように，正規化座標を有効にすることで，画像の大きさにかかわらず 0～1 の範囲でアクセスできることが分かる．これにより，スケールに依存しないメモリアクセスが可能となる．今回はこの機能を利用することでマルチテンプレートと同等の機能を実現する．

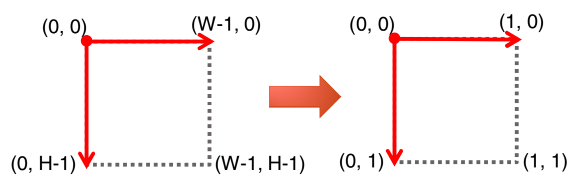

図 12　正規化座標の有効／無効による違い．（左）有効でない場合，（右）有効な場合．

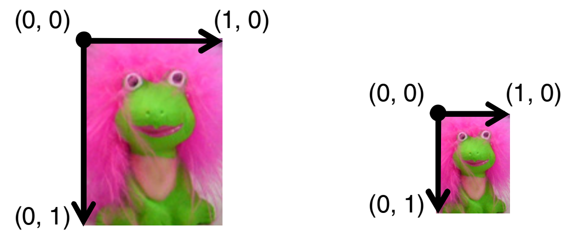

図 13　正規化座標を利用したメモリアクセス

テクスチャメモリへのアクセスは非常に簡単であり，図 11 の 19～20 行目のように tex2D にテクスチャメモリを指す変数と座標を指定する．ただし，tex2D の戻り値はテクスチャの定義（Type と ReadMode の組み合わせ）に依存するため注意が必要である．

最後に，図 11 の 4～34 行目が入力画像とテンプレート間の類似度を計算する部分である．今回は SSD (Sum of Squared Difference) を利用し，GPU 上の各スレッドがこの関数を実行する方法を採用する．ここで，各スレッドがどの位置を計算するかは 6～7 行目で決定している．これは，入力画像を図 14 右のような 16×16 のブロックで分割し，各ブロック内の各画素を GPU 上の各スレッドが計算する方法である．この方法は非常に単純ではあるが，さまざまな場面で利用できる有効な分割方法である．また，CPU からこの関数を呼び出す際は，3.1 で説明したスレッドの階層が図 14 に従うように設定する必要がある．

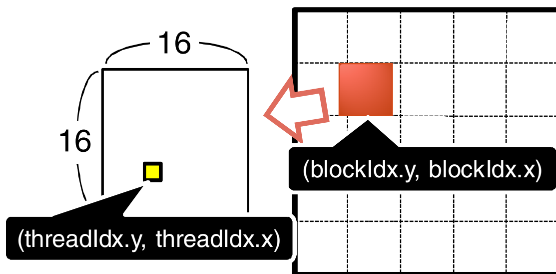

図 14　各スレッドの計算位置

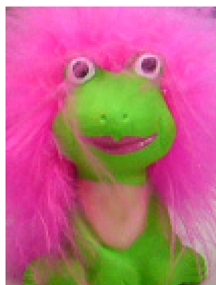

図 15　テンプレート画像

図 16　実験結果

5.3 計算速度の評価

上記で説明したプログラムを計算機上に実装し，CPU と GPU の計算速度を比較した．使用した計算機は，

CPU: Intel Core2 Quad Q9550 (2.83 GHz),
GPU: NVIDIA GeForce GTX280

である．CPU には 4 つのコアが搭載されているため，OpenMP [6] を利用して 4 スレッドで並列計算するようにプログラムを実装した．また，GPU には 30 基のマルチプロセッサ（1 基あたり 8 個のスカラープロセッサが内蔵されている）が搭載されている．

入力画像は 800×600 画素である．また，図 15 は実験に用いたテンプレート画像であり，大きさは 105×135 画素である．また，スケールに関しては 0.3～1.8 倍（拡大率 1.2）とした．GPU 上に実装したテンプレートマッチングを実行した結果を図 16 に示す．図中の枠で囲まれた部分が最もテンプレートに類似する部分であり，テンプレートと同じ物体が正しく枠で囲まれていることが分かる．また，CPU は処理に約 48.7 秒必要であったが，GPU は約 2.0 秒で処理が終了した．この結果から，GPU を利用することで約 24 倍の高速化が得られることを確認した．

6. SIFT の実装

近年，回転やスケールに対して頑健な特徴である SIFT（Scale Invariant Feature Transform）が注目を集めている[7][8]．特に，SIFT の特長を活かした画像間のマッチングや物体認識・検出に関する研究が盛んである．しかしながら，SIFT は複数スケールで DOG（Difference Of Gaussian）の計算が必要なため，実時間での処理が難しいと

いう問題があった．そこで，本節では SIFT を GPU 上に実装した場合にどの程度の高速化が得られるかを示す．本節では SIFT を GPU に実装する際の基本的な考え方の説明のみを行い，具体的な実装方法は文献[9]に譲る．

6.1 高速化の基本戦略

SIFT の実装において最も計算コストの高い処理の一つが DOG である．DOG の計算の大部分は，2 次元ガウシアンフィルタの処理に費やされる．ここで，2 次元ガウシアンフィルタは 1 次元ガウシアンフィルタの畳み込みで表現することができる．この性質を利用し，1 次元ガウシアンフィルタを X 軸方向と Y 軸方向に適用することで 2 次元ガウシアンフィルタの高速化を行う．本処理では，1 次元テクスチャを効果的に使うことでメモリアクセスの効率化を図ることが可能である．また，DOG 画像からのキーポイント検出は画素単位で行うことが可能なため，キーポイント検出は GPU を用いて容易に高速化可能である．処理の流れを図 17 にまとめる．図に示すように，SIFT を実装する際は CPU と GPU の処理をうまく切り分ける必要がある点に注意が必要である．

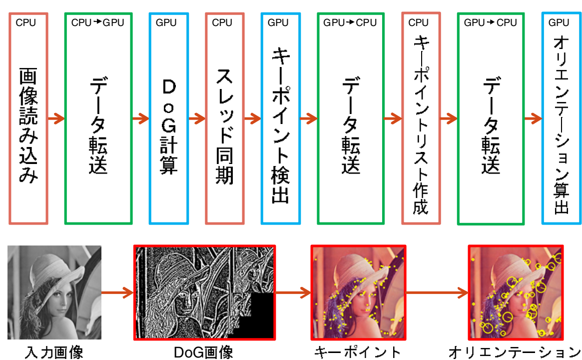

図 17　処理の流れ（SIFT）

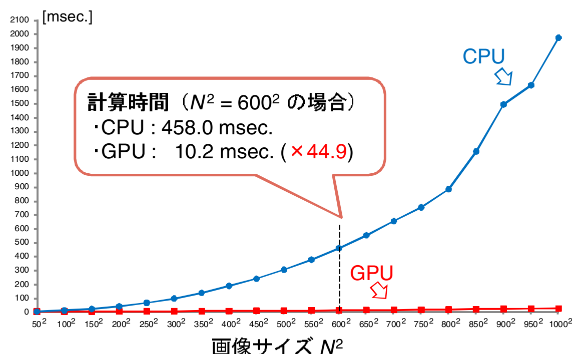

図 18　計算時間（SIFT）

6.2 計算速度の評価

5.3 で用いた計算機上に SIFT を実装し，CPU と GPU それぞれの計算時間を評価した．入力画像サイズを 50×50～1000×1000 の範囲で変化させた場合の計算時間を図 18 に示す．図から分かるように，GPU を利用することで約 45 倍（画像サイズが 600×600 の場合）の高速化が得られ，SIFT をリアルタイムで計算可能なことを確認した．今回はキーポイントの検出とオリエンテーションの算出までを GPU 上に実装したが，興味のある人は SIFT 特徴量の実装にも挑戦していただきたい．

7. むすび

本稿では，近年注目を集めている GPGPU の紹介を行った．また，開発環境として CUDA を利用し，GPGPU を非常に簡単に行えることを示した．そして，GPU を用いることで計算コストの高い画像処理アルゴリズムを容易に高速化できることを示した．しかしながら，すべての画像処理アルゴリズムを GPU で高速化できるとはかぎらない．一般的に，空間フィルタリングや局所特徴量の計算は GPU での高速化が容易であるが，ラベリングや細線化といった逐次型の画像処理アルゴリズムは高速化が困難である．また，CUDA で複数の GPU を利用する際は，各 GPU を操作する CPU スレッドを別途用意する必要がある．そのため，複数 GPU の利用には OS のスレッド管理に関する知識が少しばかり必要となる．今後，複数の GPU を簡単に扱えるようなフレームワークが登場することを期待したい．最後に，本稿を読まれた読者が GPGPU に少しでも興味を持っていただければ幸いである．

謝辞　日頃より熱心に御討論頂く名古屋大学村瀬研究室諸氏に深く感謝する．特に，プログラムの作成等で協力いただいた，名古屋大学の二村幸孝先生，野田雅文君に深く感謝する．

参考文献

[1] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-Based Visual Simulation on Graphics Hardware," Proceedings of SIGGRAPH 2002 / Eurographics Workshop on Graphics Hardware 2002, pp.1-10, 2002.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, Vol.26, No.1, pp.80-113, 2007.

[3] "CUDA Programming Guide," http://www.nvidia.com/object/cuda_develop.html

[4] "CUDA ZONE," http://www.nvidia.com/object/cuda_home.html

[5] 村瀬 洋, V. V. Vinod, "局所色情報を用いた高速物体探索--アクティブ探索法 -- (Fast visual search using focussed color matching - active search -)," 電子情報通信学会論文誌, Vol. J81-DII, No.9, pp.2035-2042, 1998.

[6] "OpenMP," http://openmp.org/

[7] David G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60, 2, pp. 91-110, 2004.

[8] 藤吉弘, "Gradient ベースの特徴抽出 -SIFT と HOG-," 情報処理学会 研究報告, CVIM 160, pp. 211-224, 2007.

[9] "SiftGPU: A GPU Implementation of SIFT," http://www.cs.unc.edu/~ccwu/siftgpu/