修士論文

題目

トリラテラルフィルタを用いた SIFTのGPUによる高速化

指導教員

大野　和彦

2024年

三重大学大学院 工学研究科 情報工学専攻
コンピュータアーキテクチャ研究室

澤田　裕貴 (420M505)

トリラテラルフィルタを用いたSIFTのGPUによる高速化

澤田　裕貴

内容梗概

近年、ドローンによる空撮や物体測量など多視点からの撮影画像を利用し、3 次元形状を復元する Structure from Motion(SfM) が用いられてきている。SfM の計算には Scale-Invariant Feature Transform (SIFT) と呼ばれる特徴量抽出アルゴリズムが使用されており、SIFT 内で使用するフィルタとしてトリラテラルフィルタを採用することで精度が向上するが、計算量が増加し実行速度が低下する。

そこで本研究では、トリラテラルフィルタを用いた SIFT アルゴリズムを GPU 上に実装することで高速化する手法を提案する。提案手法では、トリラテラルフィルタの内部処理を画素ごとに並列化する。この内部処理の内訳として、Detail Bilateral Filter 関数の計算が大きな比重を占めている。この関数は一般的なフィルタと同様に画素ごとにその近傍画素を参照して計算を行うが、画素ごとにウィンドウサイズが可変であり計算量が均質でないという特徴を持つ。このため、GPU 上で単純に画素ごとの並列化を行うと計算量の異なるスレッドを SIMD 型で同時実行することになり、一部の計算コアがアイドルとなって実行速度が低下する。そこで提案手法では、スレッドの担当画素をウィンドウサイズでソートすることにより同時実行するスレッドの計算コストを均一化し、実行効率を向上させる。

性能評価を行うため、Detail Bilateral Filter を CPU、GPU に実装し、さらに GPU 上で提案手法を適用した実装を行なった。これらを用いて実行時間を測定した結果、GPU 版は CPU 版に対して 24〜95 倍の速度向上が得られた。また GPU 版に提案手法を適用することで、さらに 1〜1.27 倍の速度向上が得られた。

一方で、GPU とトリラテラルフィルタの使用が SIFT 計算に与える影響を評価するため、ガウスフィルタを使用する CPU 版、トリラテラルフィルタを使用する CPU 版及び GPU 版の 3 種類の SIFT 計算プログラムを実装し、結果の精度とマッチング数を比較した。その結果、いずれの実装でも高精度の対応結果が得られており、CPU 版と GPU 版でもトリラテラルフィルタを用いた場合の差異はなかった。またガウスフィルタと

三重大学大学院　工学研究科

比較してトリラテラルフィルタを用いた場合はマッチングした特徴点の個数が 3.4 倍に増えており、後者の優位性が示された。

Accelerate Trilateral Filter SIFT algorithm on GPU

Sawada Yuki

Abstract

In recent years, Structure from Motion (SfM) has been used to reconstruct three-dimensional shapes using images taken from multiple viewpoints, such as aerial photography by drones and object surveys. A feature extraction algorithm called Scale-Invariant Feature Transform (SIFT) is used to calculate SfM, and adopting a trilateral filter as a filter used within SIFT improves accuracy, but increases the amount of calculation. The execution speed decreases.

Therefore, in this research, we propose a method to speed up the SIFT algorithm using trilateral filters by implementing it on GPU. In the proposed method, the internal processing of the trilateral filter is parallelized for each pixel. As a breakdown of this internal processing, the calculation of the Detail Bilateral Filter function occupies a large proportion. This function, like a general filter, performs calculations for each pixel by referring to its neighboring pixels, but has the characteristic that the window size is variable for each pixel and the amount of calculation is not uniform. For this reason, if you simply perform pixel-by-pixel parallelization on the GPU, threads with different calculation amounts will be executed simultaneously in SIMD type, and some calculation cores will become idle and the execution speed will decrease. Therefore, the proposed method equalizes the calculation cost of concurrently executing threads and improves execution efficiency by sorting the pixels assigned to each thread by window size.

In order to evaluate the performance, we implemented the Detail Bilateral Filter on the CPU and GPU, and also applied the proposed method on the GPU. As a result of measuring execution time using these, the GPU version was 24 to 95 times faster than the CPU version. Furthermore, by applying the proposed method to the GPU version, a further speedup of 1 to 1.27 times was obtained.

On the other hand, in order to evaluate the influence of the use of GPU and trilateral filters on SIFT calculations, we implemented three types

三重大学大学院　工学研究科

of SIFT calculation programs: a CPU version using a Gaussian filter, a CPU version using a trilateral filter, and a GPU version.

We compared the accuracy and number of matching results. As a result, high-accuracy results were obtained with both implementations, and there was no difference between the CPU and GPU versions when using trilateral filters. Furthermore, when using a trilateral filter compared to a Gaussian filter, the number of matched feature points increased by 3.4 times, demonstrating the superiority of the latter.

目 次

図 目 次

表 目 次

1　はじめに

近年ドローンや物体測量の分野において多視点から撮影した画像から 3 次元物体を復元する技術である Structure-from-Motion が注目され広く用いられてきている。SfM の計算処理では特徴量を利用しており、その抽出には Scale-Invariant Feature Transform (SIFT) が用いられている。SFT アルゴリズムは画像の回転やスケーリング、ノイズや照明といったものにも堅牢な特徴点を提供する。SIFT の計算では精度の向上が課題の一つとなっており内部計算のフィルタを置き換えることで高精度化を実現した研究がなされている。しかし高精度化と引き換えに速度の低下が課題となっていた。そこで我々は SIFT アルゴリズムの高精度化を GPU で行い高速化する手法を提案する。

2　背景

2.1　SfM

Structure-from-Motion(SfM) とはある物体を多視点から撮影した画像から、対象の物体の 3 次元形状を復元する技術である [2]。図 2.1 のようにそれぞれの画像からは特徴量というものを抽出することが可能で、異なる角度から撮影した画像から同一点の特定を行う。画像ごとの特徴量

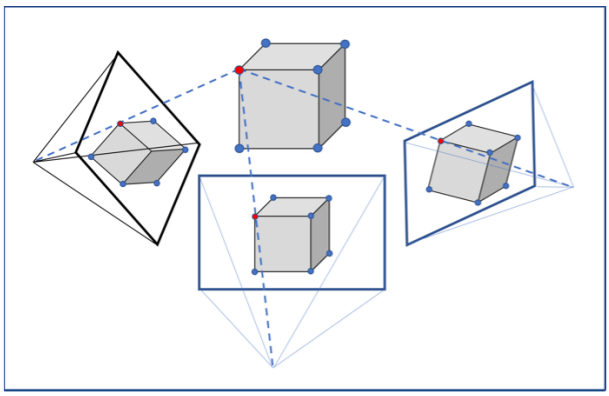

図 2.1: SfM の概要

をマッチングすることにより写真を撮影した際のカメラの位置を推定し、その後 3 次元の点群データを生成する。

2.2 SIFT

Scale-Invariant Feature Transform(SIFT) は局所特徴量を抽出するアルゴリズムである [1]。SIFT 特徴量は一つの特徴量につき 128 次元ベクトルを持つ。SIFT アルゴリズムの特徴量抽出は特徴量の検出と特徴量の記述の 2 段階で実行される。特徴量の検出では Difference-of-Gaussian 処理、極値検出、無効な特徴量の削除を行う。特徴量の記述では各特徴量の方向を表すオリエンテーション算出とヒストグラムの作成を行う。

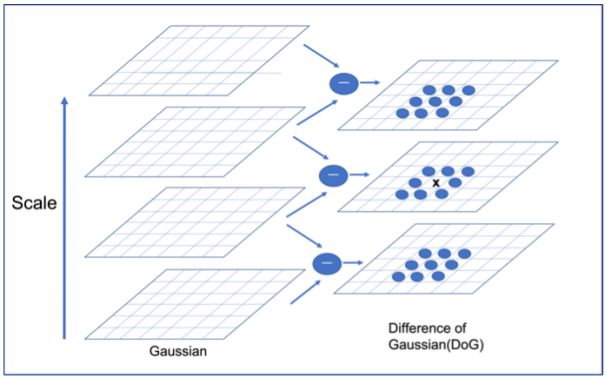

図 2.2: DoG

2.3 DoG フィルタ

Difference-of-Gaussian(DoG) とはガウスぼかしの度合いが異なる 2 つの画像の差分を取ることである。SIFT では入力画像にガウシアン処理のぼかしを徐々に入れた複数の平滑化画像を用意する。隣り合った平滑化画像の差分である DoG 画像を求める処理をする。図 2.2 のように生成された DoG 画像においてある点 x を定めた時、x と近傍 26 点の間でモノクロ画素値の大きさの比較を行う。x が近傍で極小もしくは極大だった場合、x を特徴量の候補とする。

次に特徴量の候補からノイズの影響を受けやすい点やエッジ上の点を除外する。そして検出された特徴量のオリエンテーションを求める。最後にオリエンテーションで算出された勾配強度 m と勾配方向 θ を用いて、ヒストグラムを作成する。この計算により 128 次元の特徴量を作成する。

2.4 GPU と CUDA

2.4.1 GPU

GPU(Graphic Processing Unit) はグラフィックの処理に用いられてきた。GPU の並列性を用いて汎用的な計算をする事を GPGPU(General-purpose Computing on Graphic Processing Unit) と呼ぶ。GPU は何千ものコアを搭載しており、複数の計算を並行して実行できる。GPU と CPU はそれぞれ独立したメモリを持っており、CPU から GPU にデータを転送して、GPU 上のメモリデータを用いて計算を行う (図 2.3)。GPU にはシェアードメモリと呼ばれる高速・小容量のメモリが搭載されており高速化に寄与する。コアレスアクセスと呼ばれる連続したデータアクセスにより高速なデータの読み書きが可能にもなる (図 2.4)。

2.4.2 CUDA

CUDA とは NVIDIA が自身の GPU を GPGPU として利用できるように提供している統合開発環境のことである [3]。ユーザーは C/C++を拡張した言語で記述することで並列ソフトウェアを作成することが可能である。GPU は自身のデバイスメモリのデータにしかアクセスできないため、CUDA では CPU から直接アクセスできるホストメモリから GPU の

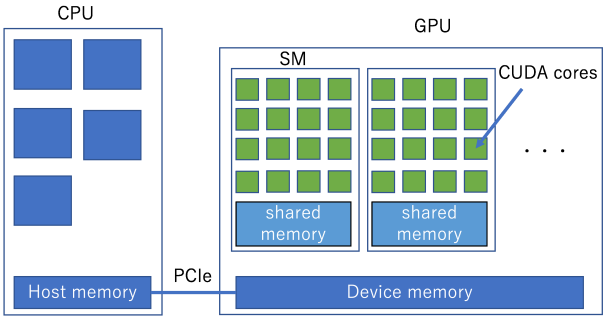

図 2.3: GPU のアーキテクチャ

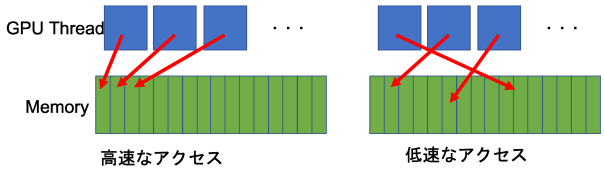

図 2.4: GPU によるメモリアクセス

みがアクセスできるデバイスメモリへのコピーを明示的に記述する必要がある。

2.5 SIFT の課題

2.5.1 計算量

SIFT は計算量が多いという問題が存在する。安倍らの研究 [4] では 640×640 の画像から 3762 個の特徴量を CPU で計算した際、Intel Core2 Duo, 2.66GHz で約 2 秒の計算時間がかかっている。画像サイズが大きくなるに従い計算量も増える。

2.5.2 開口問題

特徴点の対応をとる際に、エッジ上に抽出された点は互いに似たパターンになる可能性がある。これによりどの点と対応させるのが正解か判断できない問題が生じる。これが開口問題と呼ばれている。

2.6 GPU による SIFT 高速化

SIFT 演算を高速化する研究は過去に行われてきている。また Wu らによるオープンソースの SiftGPU[5] は GPU で SIFT の高速化を達成している。Zhihao らの研究 [6] では GPU 上での SIFT 演算を最適化することにより、OpenCV の実装より約 59 倍に高速化した事例もある。Mahdi らによる研究 [7] ではネットワークに接続されたコンピュータで実行する分散並列コンピューティングにより約 1.9 倍の高速化を達成している。

2.7 フィルタアルゴリズム置き換えによる高精度化

2.7.1 バイラテラルフィルタ

SIFT アルゴリズムはオリジナル版の他にも内部処理を置き換えることで高精度化、高速化を試みた研究が行われている。その一つが DoG フィルタで使われるガウスフィルタを置き換えるものである。ガウスフィルタをバイラテラルフィルタに置き換え GPU で実行することで高精度化と

高速化を目指した研究がある [8]。このフィルタはエッジ保存平滑化フィルタと呼ばれ、エッジ上の特徴量を排除することが可能となる。この研究では 512×512 の画像で CPU と比較して約 6 倍の高速化を達成している。

2.7.2　トリラテラルフィルタ

SIFT アルゴリズム内のフィルタをトリラテラルフィルタに置き換えることで高精度化をした研究が存在する [9]。これも 3.3.1 と同じくエッジ保存平滑化フィルタと呼ばれるもので開口問題に対処することが可能となっている。結果として特徴量の対応点数は約 2 倍に増加し誤対応点数は 1/10 となり高精度になったが CPU での実行時間が最大で 150 倍に伸びる課題が存在する。

2.8　トリラテラルフィルタ SIFT の課題

トリラテラルフィルタの特徴として画素ごとにウィンドウサイズが可変であることが挙げられる。一般的なフィルタはウィンドウサイズが固定されているため計算コストは一定である。しかしトリラテラルフィルタは例として 640×640 の画像だと最小で 1×1、最大で 129×129 の近傍画素を参照してフィルタをかける。このため計算コストが大きくなり実行時間が大幅に伸びる課題が存在している。そこで本研究では GPU を

用いることで高速化を行う。

2.9 トリラテラルフィルタのアルゴリズム

トリラテラルフィルタは入力としてフィルターをかける対象の画像、パラメーター σ_c を用いて計算を行う。内部的に多くのパラメータを必要としているが、先に挙げた σ_c を除いて全てのパラメーターを動的に決定している。以下でフィルタのアルゴリズムを説明する。

2.9.1 パラメーター決定 1

$\sigma_{c\theta} = \sigma_c, beta = 0.15, Filtersize = \sigma_c$, 画像の縦横の長さから $levelMax$ というパラメーターを決定する。

2.9.2 ComputeGradients

オリジナル画像から画像勾配を計算する。画像勾配は隣接する画素の X 方向 Y 方向それぞれの値の差のことである。この計算ステップで X 勾配画像と Y 勾配画像を求める。

2.9.3 buildMinMaxImageStack

先の計算で求めた X 勾配画像と Y 勾配画像, パラメータ $levelMax, beta$ を用いて最小勾配と最大勾配、σ_s を求める。

2.9.4　パラメーター決定 2

$\sigma_{s\theta} = R = \sigma_s$ とパラメーターを決定する。

2.9.5　BilateralGradientFilter

X 勾配画像と Y 勾配画像に対してバイラテラルフィルタを適用する。この際パラメーターとして $\sigma_c, \sigma_s, Filtersize$ を用いる。これによって平滑化 X 勾配画像と平滑化 X 勾配画像を生成する。

2.9.6　findAdaptiveRegion

最小勾配と最大勾配、パラメーター R,$levelMax$ を用いて各画素ごとの f_θ を求める。

2.9.7　DetailBilateralFilter

オリジナル画像に対してバイラテラルフィルタを適用する。パラメーターとして平滑化 X 勾配画像, 平滑化 Y 勾配画像, $f_\theta, \sigma_{c\theta}, \sigma_{s\theta}$ を用いる。これによりトリラテラルフィルタが適用された画像を生成する。各画素ごとに適用されるウィンドウサイズは可変であり、2^n(n は 1 以上の整数)である。

3　提案手法

3.1　GPU を用いた並列化と計算の最適化

トリラテラルフィルタは計算負荷が高いため GPU を用いることで高速化を行う。先に提示した計算のうちパラメータの決定と buildMinMax ImageStack 以外はそれぞれ各画素ごとに画像処理を行う。そのため GPU 上では画素ごとに並列化を行うことで高速化を試みる。今回は最も計算処理の重い DetailBilateralFilter 関数の並列実装を行なった。それ以外の関数についても実装はできていないが、並列化の手法を示す。

3.1.1　ComputeGradient の並列実装

1 画素の計算を行うカーネル関数は Listing 1 のようになる。この関数を実行する GPU スレッドを画素数分生成することで、並列実行が行われる。`srcImg` は入力画像、`pX`, `pY` はそれぞれ X, Y 方向の勾配画像である。Raw2D は画像の幅、高さ、画像データの 2 次元配列を含む構造体である。`srcImg->data[i][j]` で画素値にアクセスできる。getMyX(), getMyY() はスレッド ID から担当座標を計算するマクロである。

Listing 1:

```
1 __global__ computeGradient (Raw2D *srcImg, Raw2D *pX,
      Raw2D *pY){
2   int i, j;
3   float Cval, Eval, Nval;
```

```
4
5   i = GetMyX(); j = GetMyY();
6   Cval = srcImg->data[i][j];
7   Eval = srcImg->data[i+1][j];
8   Nval = srcImg->data[i][j+1];
9   pX->data[i][j] = Eval - Cval;
10  pY->data[i][j] = Nval - Cval;
11 }
```

3.1.2 findAdaptiveRegion の並列実装

pMinStack, pMaxStack, R, levelMax を入力として、f_θ を出力する。Raw3D は Raw2D の 1 次元配列 z、Raw2D のサイズを含む構造体である。関数は画素ごとに並列実行される。Code 2 に実装を示す。

Listing 2:

```
1  __global__ findAdaptiveRegion(Raw3D *pMinStack, Raw3D *
       pMaxStack, float R, int levelMax, Raw2D *f_theta){
2   int i, j, imax, jmax, lev;
3   i = GetMyX(); j = GetMyY();
4   for(lev = 0; lev < levelMax; lev++) {
5     if (pMaxStack->z[lev]->data[i][j] >
6         (pMaxStack->z[0]->data[i][j] + R) ||
7         pMinStack->z[lev]->data[i][j] <
8         (pMaxStack->z[0]->data[i][j] - R))
9       break;
10  }
11  f_theta->data[i][j] = lev - 1;
12 }
```

3.1.3　ワープ内の計算量の均一化

トリラテラルフィルタを GPU で実装するにあたり、画素ごとに並列化をしたとしても先述した可変ウィンドウサイズによって計算時間が異なる。GPU のスレッドは 32 本 (1 ワープ) 単位で SIMD 型実行が行われるため、仕事が早く終わってしまったコアはアイドル状態となってしまう。

この問題に対処するため、GPU の各スレッドが担当する画素を負荷の軽いものから重いものへソートすることでワープ内の計算量の均一化を図る (図 3.5,3.6)。各画素のウィンドウサイズはあらかじめ計算可能なものである。そこで、ウィンドウサイズと画素の位置が入った構造体のテーブルを作り、それをウィンドウサイズでソートする (図 3.7)。

GPU のスレッドは組み込み変数を用いて自身の ID を取得し、それに従って処理を分担する。ここで GPU のスレッドに 0 から始まる連続した整数が ID として振られているとする。通常、幅 w ピクセルの画像に対して座標 (x, y) の画素は ID が $(w \times x + y)$ のスレッドが処理するが、本手法ではテーブルに格納された x, y 座標を参照してその画素のフィルタ計算を行う。

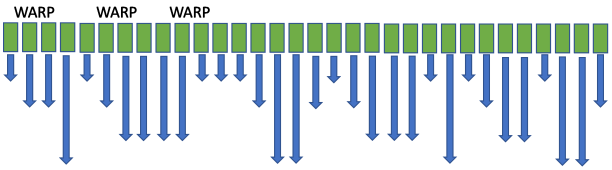

図 3.5: ソート前

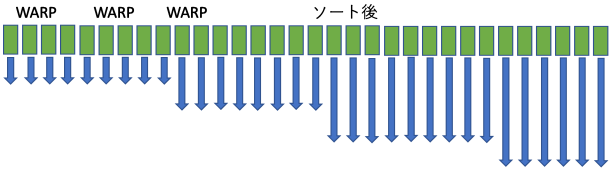

図 3.6: ソート後

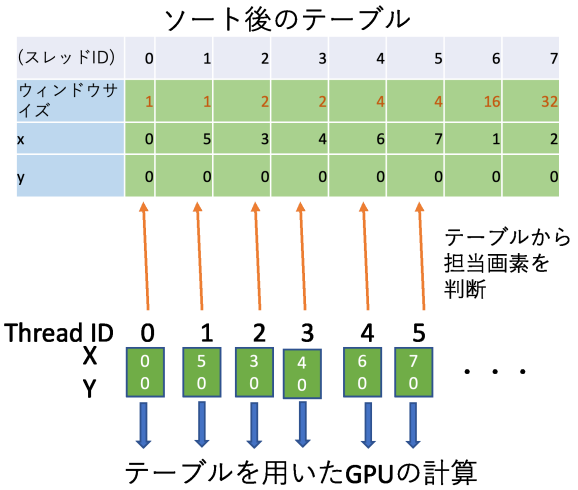

図 3.7: ソートされたテーブル

3.2　データレイアウト最適化によるメモリアクセスの高速化

GPU 上では連続したメモリにアクセスすることでより高速なデータアクセスが可能となるコアレスアクセスというものがある。GPU のスレッドは先のテーブル参照により連続したデータアクセスでは無くなってしまっている。そこでウィンドウサイズでソートを行う際に安定ソートを用いる。これにより GPU の各スレッドが近い位置の画素を参照する確率が高くなりデータアクセスの高速化が期待できる。

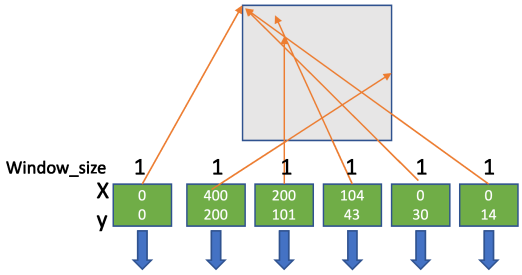

図 3.8: ソート前

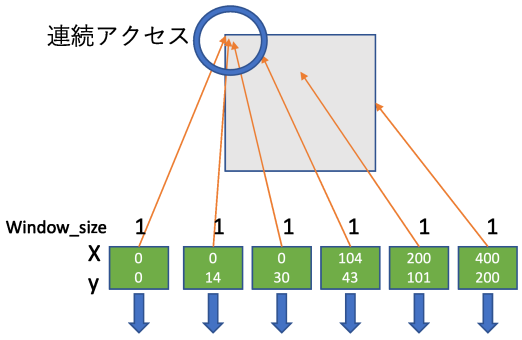

図 3.9: ソート後

3.3 各画素ごとのフィルタ計算の並列化

2.8 で述べたように各画素ごとのフィルタ処理においてウィンドウサイズが可変であることが課題であるが、それぞれの計算は独立している (Code 3)。そこでウィンドウサイズ内で参照する画素全てをさらに並列化することで高速化を図る。図 3.10 のように計算された結果を一時的に配列に格納し、全ての計算結果が配列に格納されてから、最後に全ての値の処理を行うことで高速化を行う。

Listing 3:

```
__global__ DetailBilateralFilter(int i,
                                 int j,int w){
  int m, n; float a, b;
  a = b = 0;
  for m = -w; m <= w; m++ {
    for n = -w; w <= w; w++ {
      a += calc_a(m,n); b += calc_b(m,n);
      }
    }
  a = a/b;
}
```

3.4 計算の削減

エッジ保存平滑化フィルタであるトリラテラルフィルタを用いることで DoG 画像を生成する際にエッジ付近で差分が生じないようになる。このため、SIFT 演算において本来必要であったエッジ上の点を除外する計

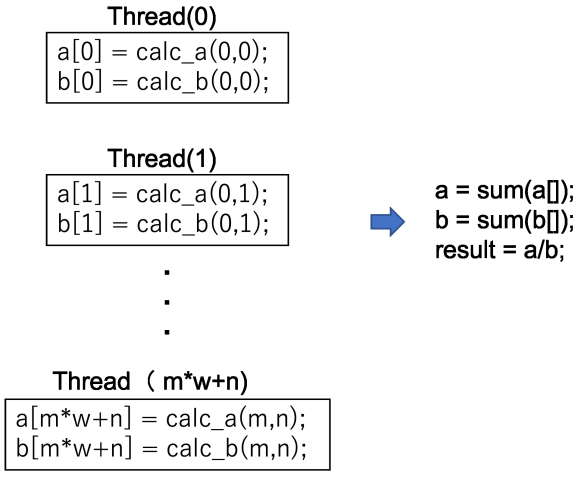

図 3.10: DetailBilateralFilter の並列化

算が不要となる。このような計算の省略を行うことで GPU での実行速度をさらに高速化する。

3.5 並列実行可能な関数の重ね合わせ

トリラテラルフィルタの計算は 5 ステップを得て完了する。このステップのうち 3 つ目のバイラテラル勾配フィルタと 4 つ目の計算は互いに独立しているため並列実行が可能である。これにより計算速度の高速化を図る。

3.6 高速な数学関数への置き換え

プログラム中での割り算といった数値計算において CUDA が提供している精度が低いがより高速な関数に置き換えることで高速化を実現する。

ただし高精度化を目的にトリラテラルフィルタを導入しているので性能低下と速度向上がどうなるか実験が必要である。

4　実験環境・評価手法

4.1　実験環境

今回はトリラテラルフィルタの中で特に処理が重い DetailBilateralFilter 関数に提案手法 (並列化、ワープ内の処理の均一化) を適用して実験を行った。実験は 3 種類行う。１つ目は CPU 版、GPU 版,GPU ソート版で DetailBilateralFilter 関数の実行時間を比較した。2 つ目は特徴点のマッチング制度を比較する。３つ目はオリジナルの SIFT, トリラテラルフィルタを用いた SIFT の CPU 版と GPU 版に対してノイズなどの外乱を付加した複数の画像を用意して、特徴点マッチング精度比較を行なった。

性能評価に用いる環境は CPU に Intel Xeon E5-2620 v2/(2-2.5GHz, 6 Core, 16G)、GPU には NVIDIA の RTX-2080(1.52 GHz, 8GB) を用いた。SIFT の実装には Open Source Software である OpenSIFT を元に、提案手法を追加実装した。

4.2　予備実験

予備実験としてトリラテラルフィルタの実行時間の内訳を計測した。

表 4.1: トリラテラルフィルタの実行時間の内訳

	100x100(ms)	160x160(ms)	320x320(ms)
ComputeGradients	0.09	0.228	0.34
buildMinMaxImageStack	0.897	3.159	11.803
BulateralGradientFilter	9.264	25.589	93.424
findAdaptiveRegion	0.163	0.419	1.917
DetailBilateralFilter	675.332	11108.6	221361

表 4.1 で示すように、DetailBilateralFilter が実行時間の大半を占めているため、この部分の高速化が重要であることがわかる。

5 実験結果

5.1 DetailBilateralFilter 関数の実行速度

実験結果としてまずトリラテラルフィルタ内部にある DetailBilateral-Filter 関数の実行速度を比較する。比較対象は CPU,GPU 実装、ソート版 GPU 実装となっている。実験結果を表 5.2 で示す

表 5.2: DetailBilateralFilter 関数の実行時間

Image size	CPU	GPU	GPU(ソート版)
160 × 160	11844.6ms	489.216ms	489.216ms
320 × 320	247773ms	4683.26ms	3779.12ms
640 × 640	4148530ms	53347.9ms	43515ms

表 5.3: 特徴点のマッチング精度

フィルター アルゴリズム	正しく マッチングした 特徴点数	マッチングした 特徴点数	精度 (%)
ガウスフィルタ (CPU)	7	7	100
トリラテラルフィルタ (CPU)	24	24	100
トリラテラルフィルタ (GPU)	24	24	100

5.2 SIFT 演算の精度

オリジナル版のガウスフィルタを用いたSIFT(CPU), トリラテラルフィルタを実装したSIFT(CPU), トリラテラルフィルタを実装したSIFT(GPU)の 3 種類で特徴点のマッチング精度を比較する。精度の比較は 100x100 の画像と、そこから切り取った画像の 2 種類を用意して、マッチングの誤差が $\pm$ 4 ピクセル以内に収まっていた場合に正しいものとする。マッチング精度の比較を表 5.3 に示す。また各フィルタでのマッチング結果を図 5.14 に示す。

5.3 SIFT における特徴点の精度比較

5.3.1 比較方法

図 5.15 に示すように、ノイズやコントラストの変化といった外乱に対して SIFT 演算がどれほどの影響があるかを評価した。

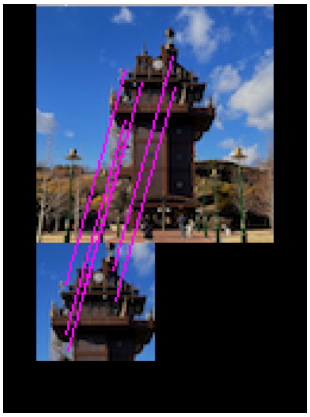

図 5.11: ガウスフィルタによる SIFT(CPU) の特徴点マッチング

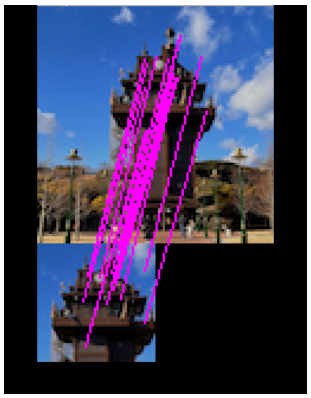

図 5.12: トリラテラルフィルタによる SIFT(CPU) の特徴点マッチング

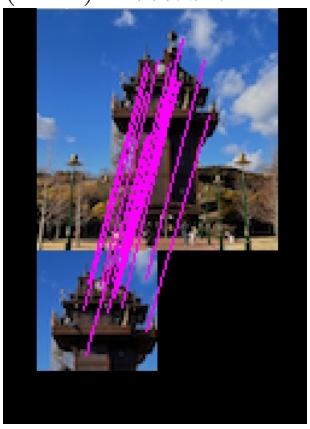

図 5.13: トリラテラルフィルタによる SIFT(GPU) の特徴点マッチング

図 5.14: 各フィルタを用いた SIFT 演算の特徴点マッチング結果

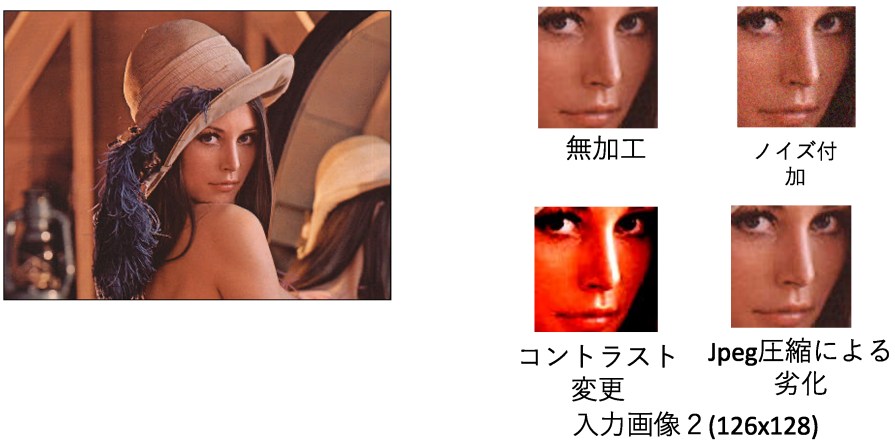

図 5.15: 元画像と外乱をつけた入力画像

5.3.2 結果

実験結果を図 6.18〜6.27, 表 6.4〜6.7 に示す。

6 考察

CPU 版と比較して実行時間が大幅に短縮されている。また、ウィンドウサイズでソートをしたことにより、通常の GPU 版に比べても 320x320、640x640 の画像で高速化されている。これは GPU 内部で適切に関数が実行されていることで無駄な待機状態のスレッドが減っているからと考えられる。ただし、全体として 1 枚の画像に対しての実行時間がガウスフィルタに比べて遅い。GPU 版は画素ごとの並列化はできているが、画素内部でのウィンドウサイズのループ処理が展開されていないため低速であ

ると考えられる。

SIFT 演算はどの結果においても精度が 100%となった。トリラテラルフィルタを用いた SIFT は CPU 版、GPU 版ともに結果は同じであるため GPU で計算した際の誤差はほぼないものと言える。また、ガウスフィルタと比べて特徴点のマッチング数が増えていることにより、より高性能になっていることがわかる。

SIFT における特徴点の精度比較の結果からは、全体の傾向としてトリラテラルフィルタの特徴点の数が増えたことがわかる。また、無圧縮のものと jpg 圧縮においてはマッチング数が増加した。

ただしコントラストをつけたものや、ノイズにおいてはマッチング数が少なかったり精度が大幅に落ちることがわかる。この結果よりトリラテラルフィルタ SIFT は無加工や jpg 圧縮には強いが、ノイズやコントラストに弱くなる傾向がわかった。

図 6.16: ガウスフィルタ

図 6.17: トリラテラルフィルタ

図 6.18: オリジナル画像に対する SIFT のマッチング精度比較

図 6.19: ガウスフィルタ

図 6.20: トリラテラルフィルタ

図 6.21: ノイズ画像に対する SIFT のマッチング精度比較

図 6.22: ガウスフィルタ

図 6.23: トリラテラルフィルタ

図 6.24: コントラストを変化させた画像に対する SIFT のマッチング精度比較

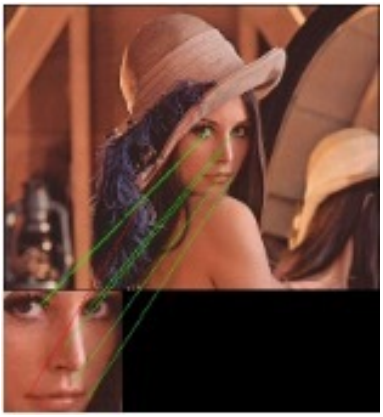

図 6.25: ガウスフィルタ

図 6.26: トリラテラルフィルタ

図 6.27: jpeg 圧縮で劣化した画像に対する SIFT のマッチング精度比較

表 6.4: 無加工の画像に対する特徴点のマッチング精度

	特徴点の数	正しくマッチングした特徴点	マッチングした特徴点	精度 (%)	実行時間 (sec)
ガウス CPU	32	7	7	100	0.320531
トリラテラル GPU	963	58	66	87.8787879	605.665184

表 6.5: ノイズの画像に対する特徴点のマッチング精度

	特徴点の数	正しくマッチングした特徴点	マッチングした特徴点	精度 (%)	実行時間 (sec)
ガウス CPU	41	3	5	60	0.347025
トリラテラル GPU	3028	2	2	100	601.922393

表 6.6: コントラストの画像に対する特徴点のマッチング精度

	特徴点の数	正しくマッチングした特徴点	マッチングした特徴点	精度 (%)	実行時間 (sec)
ガウス CPU	41	6	7	85.7142857	00.305556
トリラテラル GPU	896	46	120	38.3333333	608.582707

表 6.7: Jpeg 圧縮の画像に対する特徴点のマッチング精度

	特徴点の数	正しくマッチングした特徴点	マッチングした特徴点	精度 (%)	実行時間 (sec)
ガウス CPU	32	7	7	100	0.320531
トリラテラル GPU	963	58	66	87.8787879	605.665184

7　おわりに

本論文ではGPU上でSIFTアルゴリズムを高速化する手法を提案した。SIFTアルゴリズムの中で使われる平滑化フィルタをトリラテラルフィルタに置き換えGPUで実行することで、フィルタ内の一部の処理がCPUと比較して高速化されることが示された。SIFT演算においては高い精度を維持したままマッチングする特徴点数が増えたことで性能が向上することが示された。ただし画像の劣化や外乱のあるものでは精度が低下することもわかった。今後の課題としてフィルタ内部の全ての関数をGPUで実装し、SIFTアプリケーションに組み込むことで実際の性能評価を行うことが挙げられる。

謝辞

本研究にあたり日頃からご指導、ご助言くださいました大野和彦講師、高木一義教授、深澤祐樹研究員に感謝いたします。また研究に協力していただいた計算機アーキテクチャ研究室の方々に感謝します。

参考文献

[1] D.G. Lowe, “Distinctive image features from scale- invariant keypoints”, Intl. J. Comput. Vis. 60 (2) (2004) 91–110.

[2] M.J. Westoby, J. Brasington, N.F. Glasser, M.J. Hambrey, J.M. Reynolds, ‘ Structure-from-Motion ’ photogrammetry: A low-cost, effective tool for geoscience applications,Geomorphology,Volume 179,2012,Pages 300-314,

[3] https://docs.nvidia.com/cuda/

[4] 安倍,“高速かつメモリ消費量の少ない局所特徴量”, MIRU, vol.2011, pp.1682-1689

[5] C. Wu,“Siftgpu-v400: An mature open-source gpu implementation of sift”, 2013. http://cs.unc.edu/ ccwu.

[6] Zhihao Li, Haipeng Jia, Yunquan Zhang, “HartSift: A High-Accuracy and Real-Time SIFT Based on GPU”, 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)

[7] M. S. Mohammadi and M. Rezaeian, “SiftD: A CPU & GPU distributed hybrid system for SIFT”, 7’th International Symposium on Telecommunications (IST’2014), Tehran, Iran, 2014, pp. 613-618

[8] 山崎，甲藤, “Bilateral Filter を用いた SIFT の性能改善”，信学技報, vol. 109, no. 469, IE2009-184, pp. 29-34, 2010 年 3 月.

[9] 水野,“トリラテラルフィルタの SIFT への適用”,ISS 2015 年 総合大会

A　付属資料

Listing 4: Filter Program Code

```
#include <thrust/reduce.h>
#include <thrust/execution_policy.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

#include "trilateral_cuda.h"

#include <stdio.h>

typedef struct
{
    int index;
    int x;
    int y;
    int m;
    int n;
    int window_size;
} map_t;

__device__ typedef struct {
    int y;
    int x;
} point_t;


__global__ void test_cuda()
{
    printf("hello world from cuda\n");
}

void test()
{
    test_cuda<<<1, 1>>>();
}

uint64_t divRoundUp(uint64_t value, uint64_t radix)
{
    return (value + radix - 1) / radix;
```

```
39  }
40
41  #define IMUL(X, Y) __mul24(X, Y)
42  #define M_EXP 2.7182818284590452353602874713527
43
44
45  __global__ void HalfSize_CUDA(float *fTheta, int *
        half_size, point_t *point, int width, int height){
46      int x = blockIdx.x * blockDim.x + threadIdx.x;
47      int y = blockIdx.y * blockDim.y + threadIdx.y;
48      int index = x + y*width;
49
50      if (x >= width || y >= height){
51          return;
52      }
53      half_size[index] = (int)pow(2.0,(int)fTheta[index])
            /2;
54      point[index].x = x;
55      point[index].y = y;
56  }
57
58
59  __global__ void WindowSize_CUDA(float *fTheta, uint64_t
        *window_size, map_t *map, int width, int height)
60  {
61      int x = blockIdx.x * blockDim.x + threadIdx.x;
62      int y = blockIdx.y * blockDim.y + threadIdx.y;
63      int index = x + y * width;
64
65      if (x >= width || y >= height)
66      {
67          return;
68      }
69      window_size[index] = (int)(pow(2.0, (int)fTheta[
            index]) / 2) * 2 + 1;
70  }
71
72  __global__ void DetailBilateralFilter_CUDA2(float *src,
        float *xSmoothGradient, float *ySmoothGradient,
        float *fTheta,float sigmaCTheta, float sigmaRTheta,
        float *dst, int width, int height, float *tmp, float
         *normFactor, uint64_t *window_size)
```

```
73  {
74
75      int x = blockIdx.y;
76      int y = blockIdx.z;
77
78      int index = x + y * width;
79      if (x >= width || y >= height)
80      {
81          return;
82      }
83      int halfSize = fTheta[index];
84      halfSize = (int)pow(2.0, halfSize) / 2;
85
86      int mn_id = blockIdx.x * blockDim.x + threadIdx.x;
87
88      if (mn_id >= window_size[index] * window_size[index
            ])
89      {
90          return;
91      }
92      int m = (mn_id) % window_size[index] - halfSize;
93      int n = (mn_id) / window_size[index] - halfSize;
94
95      if (0 <= x + m && x + m < width && 0 <= y + n && y
            + n < height)
96      {
97
98          if (index == 0)
99          {
100             printf("{%d}{%d %d}", halfSize, m, n);
101         }
102
103         float diff, detail, domainWeight, rangeWeight;
104
105         float coeffA = xSmoothGradient[index];
106         float coeffB = ySmoothGradient[index];
107         float coeffC = src[index];
108
109         diff = (float)(m * m * n * n);
110         domainWeight = (float)pow(M_EXP, (double)(-diff
                / (2 * sigmaCTheta * sigmaCTheta)));
```

```
111             detail = (float)(src[x + m + (y + n) * width] -
                    coeffA * m - coeffB * n - coeffC);
112             rangeWeight = (float)pow(M_EXP, (double)(-(
                    detail * detail) / (2 * sigmaRTheta *
                    sigmaRTheta)));
113
114
115             tmp[index * 512 * 512 + mn_id] = detail *
                    domainWeight * rangeWeight;
116             normFactor[index * 512 * 512 + mn_id] =
                    domainWeight * rangeWeight;
117         }
118         else
119         {
120             return;
121         }
122     }
123
124
125     __global__ void DetailBilateralFilter_CUDA(float *src,
        float *xSmoothGradient, float *ySmoothGradient,
        float *fTheta,float sigmaCTheta, float sigmaRTheta,
        float *dst, int width, int height)
126     {
127         int x = blockIdx.x * blockDim.x + threadIdx.x;
128         int y = blockIdx.y * blockDim.y + threadIdx.y;
129         int index = x + y * width;
130
131         if (x >= width || y >= height)
132         {
133             return;
134         }
135
136
137         float diff, detail, domainWeight, rangeWeight;
138         float normFactor = 0.0;
139         float tmp = 0.0;
140
141         int halfSize = (int)fTheta[index];
142         halfSize = (int)(pow(2.0, halfSize) / 2);
143
144         float coeffA = xSmoothGradient[index];
```

```
145         float coeffB = ySmoothGradient[index];
146         float coeffC = src[index];
147 
148         int m = 0, n = 0;
149 
150         for (n = -halfSize; n <= halfSize; n++)
151         {
152             for (m = -halfSize; m <= halfSize; m++)
153             {
154                 diff = (float)(m * m + n * n);
155                 domainWeight = (float)pow(M_EXP, (double)(-
                        diff / (2 * sigmaCTheta * sigmaCTheta)))
                        ;
156                 if (0 <= (x + m) && (x + m) < width && 0
                        <= (y + n) && (y + n) < height)
157                 {
158                     detail = (float)(src[x + m + (y + n) *
                            width] - coeffA * m - coeffB * n -
                            coeffC);
159                     rangeWeight = (float)pow(M_EXP, (double)
                            (-(detail * detail) / (2 *
                            sigmaRTheta * sigmaRTheta)));
160                     tmp += detail * domainWeight *
                            rangeWeight;
161                     normFactor += domainWeight * rangeWeight
                            ;
162                 }
163             }
164         }
165         tmp = tmp / normFactor;
166         tmp += coeffC;
167         dst[index] = tmp;
168 
169         if(index%width == 0){
170             printf("!");
171         }
172 
173     }
174 
175 
176     __global__ void DetailBilateralFilter_Sort_CUDA(float *
            src, float *xSmoothGradient, float *ySmoothGradient,
```

```
 float *fTheta, float sigmaCTheta, float sigmaRTheta
, float *dst, int width, int height, point_t *point,
 int *half_size){
177     int x = blockIdx.x * blockDim.x + threadIdx.x;
178     int y = blockIdx.y * blockDim.y + threadIdx.y;
179     int index = x + y*width;
180
181     x = point[index].x;
182     y = point[index].y;
183
184     if (x >=width || y >= height){
185         return;
186     }
187
188     float diff, detail, domainWeight, rangeWeight;
189     float normFactor = 0.0;
190     float tmp = 0.0;
191     int halfSize = half_size[index];
192
193     float coeffA = xSmoothGradient[x + y*width];
194     float coeffB = ySmoothGradient[x + y*width];
195     float coeffC = src[x + y*width];
196
197     int m = 0, n = 0;
198
199     for(n = -halfSize; n<=halfSize; n++) {
200         for (m = -halfSize; m<=halfSize; m++) {
201             if (0 <= x+m && x+m < width && 0 <= y+n &&
                 y+n < height) {
202                 diff = (float)(m*m+n*n);
203                 domainWeight = (float) pow(M_EXP, (
                     double) (-diff/(2*sigmaCTheta*
                     sigmaCTheta)));
204                 detail=(float) (src[x+m+(y+n)*width] -
                     coeffA*m - coeffB*n - coeffC);
205                 rangeWeight = (float) pow(M_EXP, (double
                     ) (-(detail*detail)/(2*sigmaRTheta*
                     sigmaRTheta)));
206                 tmp += detail*domainWeight*rangeWeight;
207                 normFactor += domainWeight*rangeWeight;
208             }
209         }
```

```
210         }
211         tmp = tmp/normFactor;
212         tmp += coeffC;
213         if(index%width == 0){
214             printf("!");
215         }
216         dst[x + y*width] = tmp;
217     }
218
219
220
221     void DetailBilateralFilter_cuda_func(float *srcImg,
            float *pSmoothX, float *pSmoothY, float *fTheta,
            float sigmaCTheta, float sigmaRTheta, float *dst,
            int width, int height)
222     {
223
224         float *src_cuda, *xSmoothGradient_cuda, *
                ySmoothGradient_cuda, *fTheta_cuda, *dst_cuda;
225         int *half_size_cuda;
226         int half_size[width*height];
227         thrust::device_vector<int> halfSize_vector_cuda(
                width*height);
228
229         uint64_t window_size[width * height];
230         point_t *point_cuda;
231         float *tmp_cuda, *normFactor_cuda;
232         printf("width:%d height:%d \n", width, height);
233
234         time_t start, stop;
235
236
237         cudaMalloc(&src_cuda, sizeof(float) * width *
                height);
238         cudaMalloc(&xSmoothGradient_cuda, sizeof(float) *
                width * height);
239         cudaMalloc(&ySmoothGradient_cuda, sizeof(float) *
                width * height);
240         cudaMalloc(&fTheta_cuda, sizeof(float) * width *
                height);
241         cudaMalloc(&dst_cuda, sizeof(float) * width *
                height);
```

```
242         cudaMalloc(&half_size_cuda, sizeof(int)*width*height
                );
243         cudaMalloc(&point_cuda, sizeof(point_t)*width*height
                );
244
245         cudaMemcpy(src_cuda, srcImg, sizeof(float) * width
                * height, cudaMemcpyHostToDevice);
246         cudaMemcpy(xSmoothGradient_cuda, pSmoothX, sizeof(
                float) * width * height, cudaMemcpyHostToDevice)
                ;
247         cudaMemcpy(ySmoothGradient_cuda, pSmoothY, sizeof(
                float) * width * height, cudaMemcpyHostToDevice)
                ;
248         cudaMemcpy(fTheta_cuda, fTheta, sizeof(float) *
                width * height, cudaMemcpyHostToDevice);
249
250
251         dim3 blockDim(128, 4);
252         dim3 gridDim(divRoundUp(width, blockDim.x),
                divRoundUp(height, blockDim.y));
253
254         HalfSize_CUDA<<<gridDim, blockDim>>>(fTheta_cuda,
                half_size_cuda, point_cuda, width, height);
255         cudaThreadSynchronize();
256
257         // printf("!?");
258         // WindowSize_CUDA<<<gridDim, blockDim>>>(
                fTheta_cuda, window_size_cuda, map_cuda, width,
                height);
259         thrust::stable_sort_by_key(thrust::device,
                half_size_cuda, half_size_cuda + width*height,
                point_cuda);
260         // uint64_t WindowSize_sum = thrust::reduce(thrust
                ::device, window_size_cuda, window_size_cuda +
                width * height, 0);
261         // printf("window_size sum: %llu\n", WindowSize_sum
                );
262
263         // cudaMemcpy(window_size, window_size_cuda, sizeof(
                uint64_t) * width * height,
                cudaMemcpyDeviceToHost);
264
```

```
265         start = clock();
266         // DetailBilateralFilter_CUDA<<<gridDim, blockDim
                >>>(src_cuda, xSmoothGradient_cuda,
                ySmoothGradient_cuda, fTheta_cuda, sigmaCTheta,
                sigmaRTheta, dst_cuda, width, height);
267         DetailBilateralFilter_Sort_CUDA<<<gridDim, blockDim
                >>>(src_cuda, xSmoothGradient_cuda,
                ySmoothGradient_cuda, fTheta_cuda, sigmaCTheta,
                sigmaRTheta, dst_cuda, width, height, point_cuda
                , half_size_cuda);
268
269
270         cudaThreadSynchronize();
271         stop = clock();
272         std::cout << "DetailBilateralFilter_CUDA time:" <<
                width << "x" << height << ":" << (double)(stop
                - start) / CLOCKS_PER_SEC * 1000 << "ms\n";
273
274         cudaMemcpy(dst, dst_cuda, sizeof(float) * width *
                height, cudaMemcpyDeviceToHost);
275
276
277         cudaFree(src_cuda);
278         cudaFree(xSmoothGradient_cuda);
279         cudaFree(ySmoothGradient_cuda);
280         cudaFree(fTheta_cuda);
281         cudaFree(dst_cuda);
282         cudaFree(half_size_cuda);
283         cudaFree(point_cuda);
284     }
```

Listing 5: Filter Header Code

```
1   uint64_t divRoundUp(uint64_t value, uint64_t radix);
2   extern void DetailBilateralFilter_cuda_func(float *
        srcImg, float *pSmoothX, float *pSmoothY,
3               float *fTheta, float sigmaCTheta, float
                    sigmaRTheta, float *dst, int width, int
                    height);
```