Vol.2017-HPC-161 No.11
2017/9/20

ライトフィールドカメラにおける GPGPU を用いた 画像処理の高速化の検討

並木 美太郎[†1]　高木 康博[†2]　本田 舜[†1]　山口 祐太[†2]

概要：計算により常に焦点を合わせ続けることできるライトフィールドカメラの研究開発では、視差画像の生成、シフト加算、小領域画像の焦点検出など多量の計算量を必要とする画像を処理する。本研究では、フル HD 解像度(1920x1080)の画像を 60fps で処理する高速カメラの画像処理を GPGPU で高速化する方式を検討し、並列化方式、メモリ配置、などを考察した結果、従来 3 秒程度かかっていた 640x480 の画像処理が、GPGPU1 台で 12ms 程度で処理できた。本稿では、高速化方式の検討結果とマルチ GPGPU での高速化の検討について述べる。

キーワード：ライトフィールドカメラ、GPGPU

1. はじめに

近年の人工知能などでは、画像を扱うことからカメラとその画像処理はますます重要になっている。筆者らは次世代の画像入力システムとして、ライトフィールド理論とその理論に基づくカメラを用いて、機械的な機構を用いずに常に焦点を合わせ続ける手法を研究している。本理論では、計算によって焦点合わせを行う。この焦点合せをリフォーカス処理と呼ぶ。この手法では、撮像後にも、リフォーカスも可能なことから、より詳細な画像解析や認識に応用できる。しかし、多量な計算量が必要なことから、高精細画像や動画像への適応は困難であった。

本研究では、ライトフィールドカメラでの画像処理を GPGPU によって高速化する。GPGPU を 1 台用いて、リフォーカス処理を行い、CPU と比較して 250 倍程度の高速化を確認した。また、マルチ GPGPU を用いた高速化を検討し、4 台の GPGPU で 60fps(frame per scond)での動画像処理が可能なことを確認した。次に高速化方式について示す

2. ライトフィールドカメラの原理

ライトフィールドカメラ[1][2]は、図１に示すように、結像レンズ、マイクロレンズアレイ、イメージセンサで構成される。結像レンズは、物体をマイクロレンズアレイの近傍に結像する。マイクロレンズアレイのそれぞれのマイクロレンズにイメージセンサの２次元画素群が対応している。

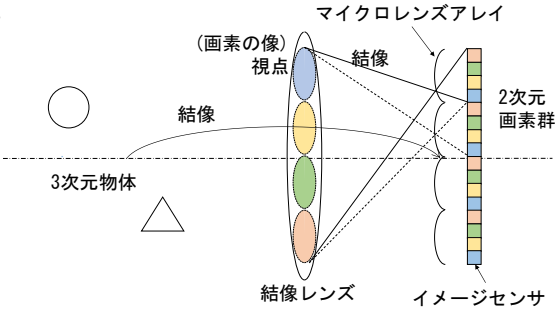

図１　ライトフィールドカメラの構造

各マイクロレンズは、結像レンズをおれぞれの２次元が祖群に結像し、結像レンズとすべての２次元画素群の間で結像関係が成り立つ。ここで、結像レンズ内に出来る各画素の像を考えて、これを視点と呼ぶ。

すべての２次元画素群の同一位置にある画素を集めて画像を構成すると、結像レンズの内の対応する視点から３次元物体を見た視差を持つ視差画像が得られる。各マイクロレンズに 2×2 画素が対応する例を図２に示す。

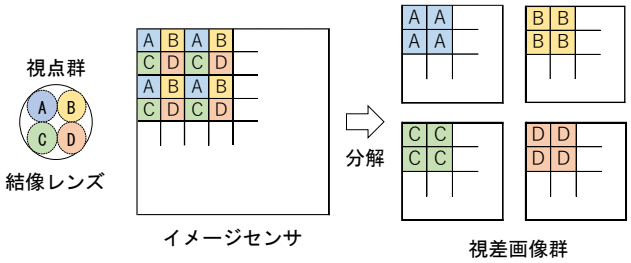

図２　要素画素群と主レンズの視点

この視差画像群からシフトしながら重ね合わせる(シフト加算と呼ぶ)ことでリフォーカス処理を行う。すべての視差画像を、対応する結像レンズ内での視点位置とリフォーカス位置に応じて、適切に縦横にシフトして加算することで、リフォーカス位置にある部分は一致してシャープになる。リフォーカス位置にない部分は一致しないため、ぼけが生じる。したがって、図３のように、リフォーカス位置に焦点があった画像が得られる。

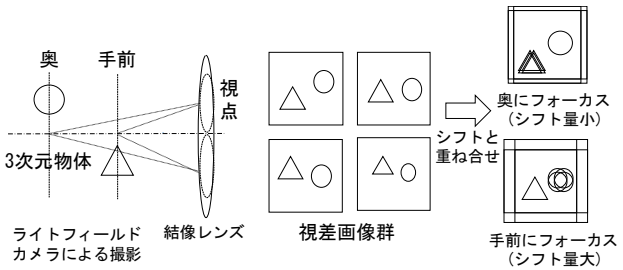

図３　リフォーカスの原理

†1 東京農工大学 工学府 情報工学専攻
†2 東京農工大学 工学府 電気電子工学専攻

Lytro 社のライトフィールドカメラ[3][4]は図 4 に示すように、複数のマイクロレンズの画像から構成される。7728×5368 ピクセル(1 画素 10bit)の画像を拡大すると、ハニカム構造のマイクロレンズからの画像を判別できる。この原画像のデータは約 50MB 程度である。この図 4 の画像から図 5 に示すような視差画像群をプログラムで生成する。このカメラでは N=15 なので、視差画像の枚数は N^2=225 枚となるが、ハニカム構造なので実際の枚数は 2×(7+11+12+13)+7×15= 191 枚である。

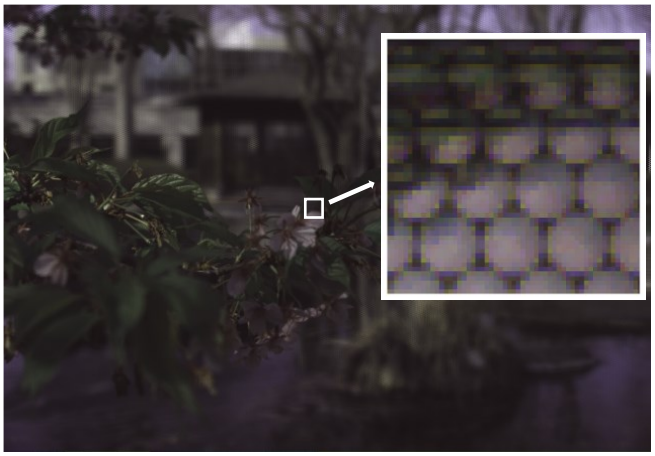

図 4　Lytro 社のライトフィールドカメラ Illum の原画像

図 5　視差画像群(640×640 画像が 15×15 枚)

ライトフィールドカメラは、撮影時に機械的な機構でのフォーカスが不要であり即時性に優れていること、また、撮影後にも画像処理により適切なフォーカス処理も可能である。たとえば、シフト量を変えることにより、ピント調整できるほか、全画面で適切なフォーカスを選択することもソフトウェア的に可能である。これらは静止画のみならず動画像でさらに優位な方式となりえる。

しかし、リフォーカス処理は多数の視差画像のシフトと重ね合せが必要となり、通常の CPU では計算時間を要する。先ほどの Lytro 社のカメラでは撮影後 PC(Personal Computer)で稼動するソフトウェアで画像処理を行う。しかし、一枚当たり、数十秒の実行時間を要する。この処理を GPGPU により高速化する。

3.　ライトフィールドカメラでの画像処理

本研究では、最終的には 4K 画像の解像度(3840×2080 ピクセル)のカメラを用いてリアルタイムな 1/60 秒のフレーム時間内で実行することを目標とする。そのために、まず、640×480 で 1/60 の処理の高速化を検討した。本研究での処理内容を図 6 に示す。

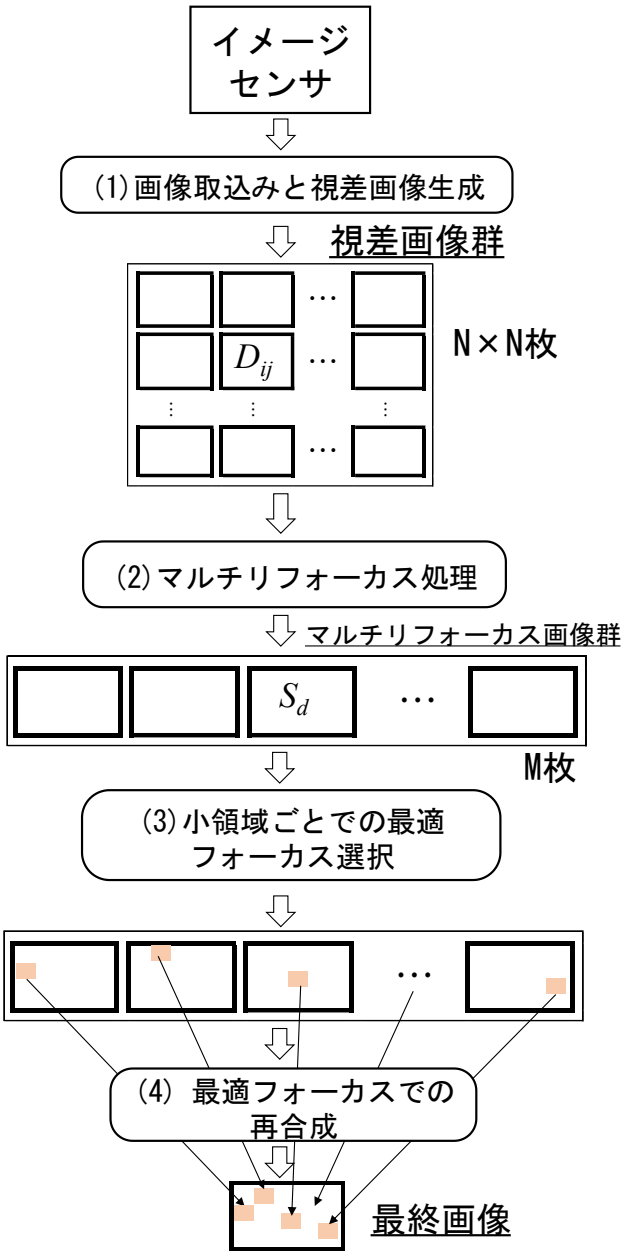

図 6　高速リフォーカス処理の全体手順

(1) イメージセンサから画像取込みと視差画像の生成

イメージセンサからのデータをマイクロレンズアレイの間の位置ずれや傾きの補正処理を行いながら、視差画像を生成する。この処理については、共同研究者が光学系を含めて現在研究中である。本稿では、図 5 に示すような

Lytro 社のカメラから生成された視差画像を用いる。先の Lytro 社のカメラでは、625×434 ピクセル、30bit/ピクセルのカラー視差画像が生成されるので、その画像を 640×480、24bit/ピクセルファイルとして格納して利用した。

(2) シフト加算によるマルチリフォーカス処理

主たる計算はこの処理に費やされる。N^2 枚の視差画像からシフト量を変えたシフト加算を行い、M 枚のマルチリフォーカス画像群を生成する。マルチフォーカスのシフト加算では、シフト量を d (非整数)、画像一枚の大きさを $X \times Y$ ピクセル、視差画像一枚を D_{ij} $(1 \le i,j \le N)$ として、そこに含まれる視差画像の 1 ピクセルを $P_{ij}(x,y)$ とする。このとき、シフト量 d で重ね合わせた画像のピクセルを $S_d(x,y)$ $(x \in X, y \in Y)$ とすると、

$$S_d(x,y) = \frac{\sum_i \sum_j P_{ij}(x - d \times i, y - d \times j)}{N^2} \quad \text{(式 1)}$$

で求められる。マルチリフォーカス処理で得られた M 枚の画像で手前にフォーカスあった画像を図 7 に、奥にあった画像を図 8 に示す。これらは、撮影後の視差画像を用いた処理である。

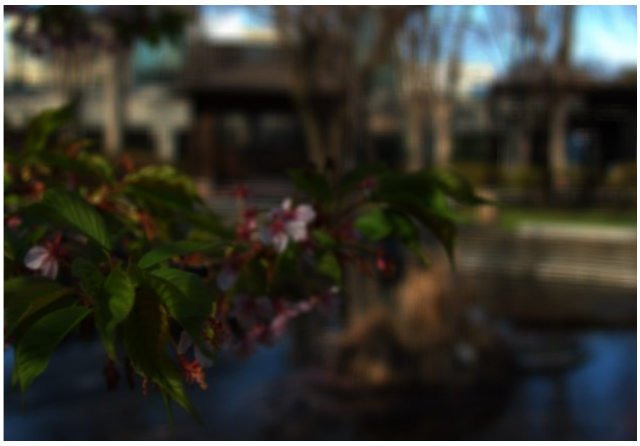

図 7　マルチリフォーカス画像の処理(手前にフォーカス)

図 8　マルチリフォーカス画像の処理(奥にフォーカス)

主記憶上のデータ量については、3 バイト/ピクセルなので、1 フレームあたりの視差画像群 N^2 枚については約 200MB である。シフト量を変えて M 枚のリフォーカス画像を生成するときの加算の計算量は、$X \times Y \times N \times N \times M$ となる。プログラム上は逐処理利は 5 重のループが出現する。本稿では、シフト量 d については、実験的にサンプル画像のフォーカスから 1/16 刻みとして、-15/16～15/16 の範囲で M=31 とした。したがって、メモリロードと加算については、640×480×15×15×31≒2G 回となる。この処理が主要項であり、ボトルネックとなっている。

(3) 小領域ごとでの最適フォーカスの選択

M 枚のリフォーカス画像について、それぞれ小領域に分割し、同じ位置にある小領域についてもっともフォーカスのあったものを選出する。シフト量 d のリフォーカス画像について、(u, v)に位置する小領域を F^d_{uv} とすると、同じ(u, v)に対して、もっともフォーカスの適切な小領域を選出する。この処理をすべての小領域に適用し、最適なフォーカスの小領域群を選ぶ(図 9)。

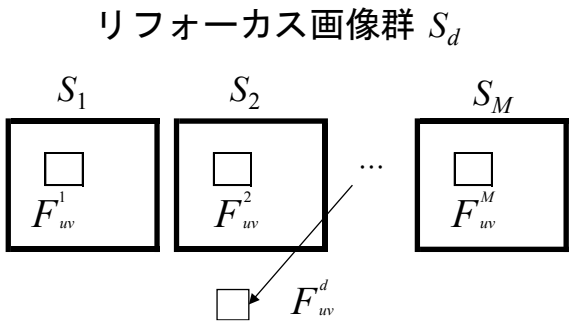

図 9　小領域ごとの最適フォーカスの選択

本稿では、小領域の大きさを 8×8 ピクセル、小領域の個数については 1 枚のリフォーカス画像について 80×60 枚の小領域とした。小領域のサイズについてはいくつか試験的に値を変えて、サンプル画像で良好なフォーカスを示すものを選んだ。M=31 の中から最適なものを一つ選ぶ処理を 80×60=4800 個のすべての小領域について行う。なお、最適なフォーカスの識別については、すべてのリフォーカス画像について輝度を生成し、$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$ のラプラシアンフィルタでエッジを抽出する。そのエッジについて、8×8 の小領域ごとに二乗和を求め、二乗和が最大の小領域を(u, v)の候補とした。

(4) 最適フォーカス画像による再合成

一つ前のステップ(3)で得られた F^d_{uv} について最適なフォーカスから得られた小領域で画像を再合成する。31 枚のリフォーカス画像から再合成した画像を図 10 に示す。

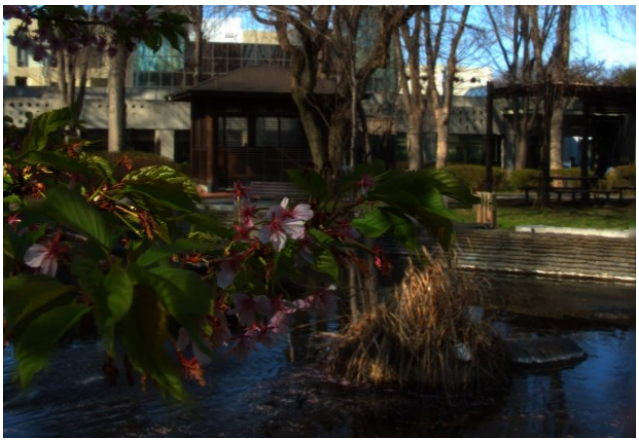

図 10 再合成後の画像

4. 逐次版での高速化

前の章で述べた処理を、まず逐次プロセッサで実現した。図 6 の(2)のシフト加算のリフォーカス処理について、定義式の式 1)をそのままコーディングした。d を固定したリフォーカス画像を一枚ずつ生成する。図 11 にアルゴリズムを、図 12 に処理を模式化した図を示す。境界などの細かい処理は省略してある。なお、計算は浮動小数点を一切使わず整数演算とした。また、加算は RGB それぞれに対して加算と平均を求めている。

```
for(d = START; d < END; ++d) {
  for(y = 0; y < Y; ++y) {
    for(x = 0; x < X; ++x) {
      v = 0;
      for(i = 0; i < N; ++i) {
        for(j = 0; j < N; ++j) {
          v += p[i][j][y-d*i][x-d*j];
        }
      }
      s[d][y][x] = v / (N*N);
    }
  }
}
```

図 11　定義式によるプログラム例(例 1)

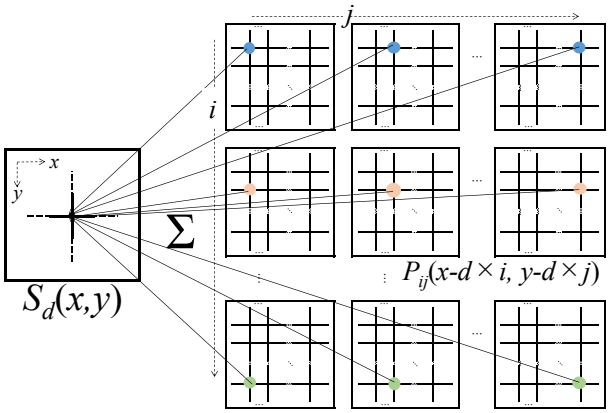

図 12 視差画像からシフト加算(プログラム例 1)

このプログラムは単純であるが、メモリのアクセス間隔が広くワーキングセットが大きくなりやすい。そこで、視差画像一枚を繰返しの内側となるように手順を変更した(図 13、図 14)。この手順では、例 1 と同様、シフト量 d を固定した上で、i と j で視差画像一枚を固定して、加算結果のピクセルをずらしながら加算する。いずれのプログラムも言語 C で 500 行程度のプログラムである。

```
for(d = START; d < END; ++d) {
  initialize s[d]
  for(i = 0; i < N; ++i) {
    for(j = 0; j < N; ++j) {
      for(y = 0; y < Y; ++y) {
        for(x = 0; x < N; ++x) {
          s[d][y+d*i][x+d*j] += p[i][j][y][x];
        }
      }
    }
  }
  for(y = 0 ; y < Y; ++y)
    for(x = 0 ; x < X; ++x)
      s[d][y][x] /= N * N;
}
```

図 13 視差画像 1 枚を内側にした逐次最適化(例 2)

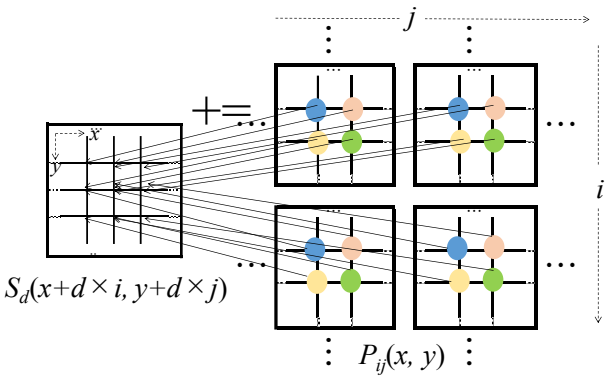

図 14 プログラム例 2 の視差画像参照と計算

シフト加算によるリフォーカス部の処理の実行結果を表 1 に示す。ファイル入出力は実行時間に含まない。1 種類の計算機で図 11 例 1 のプログラムを、3 種類の計算機で図 13 の例 2 のプログラムを実行した。コンパイラは gcc とインテル C++コンパイラを用い、最適化をしなかったときとオプションによる最適化を行った。空白部分は未計測である。性能解析ツールなどを用いた詳細な解析は未了である。

表 1 の hostA の結果を見ると、定義式 1)のプログラムを、最適化オプションなしでコンパイルすると 20 秒近くかかっていたものが、アルゴリズムを工夫した例 2 のプログラムで最適化オプションをつけると 6 倍以上の高速化が達成されている。

表１　逐次処理によるシフト加算の処理時間

	hostA		k40	gtx1080
	定義式(例1)	最適化(例2)	host(例2)	host(例2)
gcc	19.48	10.07	9.65	11.66
gcc -O	7.91	3.13	2.97	3.71
gcc -O3		3.30		
icc	7.02	3.32		
icc -O	7.02	3.32		
icc -O3		3.32		
マシン仕様	Xeon E5-1620v2@3.70GHz		Xeon E5-2637v3@3.50 GHz	Xeon E5-2650v4@2.20 GHzx2 (NUMA)

sec

表１の結果は、図６(2)のシフト加算の処理のみであった。(1)を除く(2)から(3)の全処理の実行時間を表２に示す。こちらもファイル入出力の時間は含まない。

表２　計算に関する逐次処理の全実行時間

	hostA	k40 host	gtx1080 host
(2)シフト加算	3130	2973	3710
(3)+(4) 選択と合成	37	37	73

msec，例2のプログラムをgcc -Oでコンパイル

この結果を見ると(2)のシフト加算の処理がもっとも実行時間がかかるのがわかる。また、選択と合成でも 37ms 程度かかっている。計 3 秒程度の計算時間を要することから、逐次処理では動画像は不可能である。

5. GPGPU による高速化

リアルタイムな処理については、文献[5]などがあるが、GPGPU 一つを用いて図６(2)を高速化することを考える。ライトフィールド理論を用いた GPGPU 化には文献[6]などがあるが、全体を含めた高速化を検討する。研究室には、ケイパビリティ順に、Fermi C2050、Kepler K40 で基本データを収集することとした。いくつかは、Pascal GTX 1080 で比較評価を行った。

GPGPU 一台で高速化する場合、どこを並列化するかは大きな問題である。今回実験に用いた GPGPU を表３に示す。

視差画像については、総数で 200MB 程度なので、GPGPU 上のグローバルメモリに格納できる。並列化については、スレッドは 1024 以下なので、シフト加算する 1 ピクセルごとに並列化はできない。逐次で良好な成績をおさめた図 13 の例 2 のプログラムについては、加算されるピクセルがローカルメモリに収まらないのでグローバルメモリになること、加算の際に排他制御が必要となることから、効率は良くないことが予想された。そこで、図 11 の定義式にしたがったプログラムを基本に考えることにした。基本的には図 13 の Σ は逐次の繰返しで行い、その処理を各点ごとに並列で行った。いくつかは他の並列化も実験いた。

表３　用いた GPGPU の仕様

GPGPU名	GTX1080	K40	C2050
ケイパビリティ	6.1	3.5	2.0
CUDAコア数	2560	2880	448
GPUクロック(GHz)	1.73	0.75	1.15
レジスタ数	65536	65536	32768
共有メモリサイズ(KB)	48	48	48
グローバルメモリ(MB)	11172	11520	2622
メモリクロック(MHz)	5505	3004	1500
メモリ幅(bit)	352	384	384
L2キャッシュ(KB)	2.75M	1.5M	768
リリース年	2016	2015	2011

種々の並列化を考慮したプログラムは、CUDA のブロック、グリッド指定、逐次部分、メモリ配置を換えて、50 種以上にのぼる。表４にその一部を示す。いずれも、言語 C で 500 行程度であり、nvcc –O としてコンパイルした。浮動小数点数は使っておらず、すべて整数演算である。当初は C2050 を用いていたので、その実行時間が少ないもの順に並べた。詳細な性能計測は未了である。

表４　GPGPU によるシフト加算処理の実行時間

プログラム番号	GPU逐次	ブロック	グリッド	メモリ配置	GTX1080	K40	C2050
1	15x15	640,1,1	31,480,1	d-i-y-j-x	11.8	55.4	101.0
2	15x15	640,1,1	480,31,1	d-i-j-y-x	21.9	60.6	101.0
3	10x15	64,15,1	480,31,1	d-i-j-y-x	25.3	136.0	282.0
4	10x15	64,15,1	480,31,1	d-i-y-j-x	25.0		284.0
5	15	64,15,1	10,480,31	d-i-y-j-x	25.5	137.0	289.0
6	15x15	640,1,1	480,31,1	y-x-d-i-j	23.1	410.8	
7	15x15	640,1,1	31,480,1	d-y-x-i-j	26.9	424.1	850.9
8	15x15	640,1,1	480,31,1	d-y-x-i-j	373.0	437.5	
9	32x16	32,32,1	31,1,1	d-i-j-y-x	273.8		2100.0
10	15x15	2,480,1	320,31,1	d-i-j-y-x	628.7		3500.0
11		15,15,1	640,480,31	d-i-j-y-x	158.4		6500.0

msec

もっとも高速なもので 11.8ms となっており、CPU での約 3 秒に比べて 250 倍程度の高速化が達成された。また、1/60fps でも処理可能な実行時間である。

表４については、GPU 逐次の欄は、カーネル関数内での逐次の繰返し実行の部分の回数である。ブロック、グリッドの欄は、CUDA に与える GPU 起動の際の dim 変数値である。640、480 は画像の横縦のピクセル数、15 は視差画像の横縦それぞれの枚数、31 はマルチリフォーカス画像の個数である。メモリ配置については、右の記号から連続して GPGPU のグローバルメモリに配置されていることを示す。視差画像ごとに連続した領域を考えるなら、*d-i-j-y-x* の並びとなる。つまり、*y-x* で一枚の視差画像を連続して格納し、それが *i-j* ごとに並び、マルチリフォーカス画像の添え字はもっとも外側に来ることを意味する。

表４のプログラム番号 1 と 2 の処理手順を図 15 に示す。グローバルメモリから読んで、グローバルメモリへ書き出す。ローカルメモリなどを明示的に使用していない。スレッド内の変数はすべてレジスタに割り当てられた。他のプログラムも基本構造は同じだが、カーネル関数の繰返し構造が異なっている。プログラム番号 1 と 2 では、メモリ配置とグリッドの値が異なっている。プログラム 2 は *i-j-y-x* で視差画像順にグローバルメモリに格納されているが、プログラム番号 1 は *x* のピクセルを *j* ごとに並べたものにしている。他のプログラムも基本構造は同じだが、プログラム番号 9 は別の分割の仕方、プログラム番号 11 はすべて並列で処理した。いずれも遅い。

```
// GPGPU カーネル関数
__global__ void ShiftSum {
  // x,y,d を threadIdx, blockIdx から得る
  v = 0;
  for(i = 0; i < N; ++i) { // N = 15
    for(j = 0; j < N; ++j) {
      v += P[i][j][y-d*i][x-d*j];メモリ
    }
  }
  S[d][y][x] = v / (N*N);
}

// CPU 側の GPGPU 起動
dim3 block(スレッドのパラメータ);
dim3 grid(ブロックのパラメータ);
ShiftSum<<<grid,block>>>
          (視差画像 P, マルチフィローカス画像 S の
            グローバルメモリの先頭アドレス);
```

図 15 CUDA によるシフト加算の処理
(表４のプログラム番号 1,2)

いずれの GPGPU でも、カーネル関数内で一つのマルチリフォーカス画像の 1 ピクセルを、横方向を並列で実行して生成するのがもっとも高速となった。スレッド数を有効に使えるように分割したものは、速度が低下する結果になった。

三つの GPGPU で比較した場合、並列化の方法としては同じプログラム番号 1 と 2 は、C2050 では同じ実行時間となるが、K40 で若干の差が生じ、GTX1080 では 2 倍近い差が生じている。これは、CUDA コア数、L2 キャッシュサイズが大きく影響していると考えられるが、詳細な分析は今後の課題としたい。いずれにせよ、640×480 ピクセルの解像度、24bit カラー、60fps でシフト加算は可能である。

さて、もっとも計算量を必要とするシフト加算によるマルチリフォーカス処理だけでなく、図６の(3)と(4)のフォーカスの評価、選択、合成処理も GPGPU で行った。マルチリフォーカス画像は、GPGPU のグローバルメモリ上に格納されていることから、最終結果までを GPGPU で行う。

マルチリフォーカス処理は図 15 の方法で並列化を行った。小領域ごとでの最適フォーカスの選択と最適フォーカス画像による再合成については、次の手順で GPGPU により並列処理を行った。なお、すべての結果はグローバルメモリに格納される。

(a) RGB からの輝度変換

マルチリフォーカス画像は RGB ピクセルだが、一度これを 8bit 整数の輝度に変換する。この処理は、マルチリフォーカス画像を求めるときに、同時に輝度を求めることとした。

(b) ラプラシンフィルタによるエッジ抽出

輝度からマルチリフォーカス画像 640×480 ピクセルの 31 枚内のすべての点のエッジを求める。各画像に対して、ブロック<<<640,1,1>>>、グリッド<<<480,31,1>>>で並列化。

(c) 小領域ごとに二乗和を求める

マルチリフォーカス画像について小領域ごとにエッジの二乗和を求める。8×8 の各小領域について、ブロック<<<80,1,1>>>、グリッド<<<60,31,1>>>で並列化。

(d) 最適領域の選択

エッジの二乗和が最大となる小領域を求める。31 個の小領域中の最大値を求める。ブロック<<<80,1,1>>>、グリッド<<<60,31,1>>>で並列化。

(e) 最適フォーカスの合成

最適となる小領域のマルチフォーカス画像を 31 枚から合成する。対応する *d* からの並列メモリ転送となる。ブロック<<<640,1,1>>>、グリッド<<<480,1,1>>>で並列化。

上記の(a)～(e)ごとにカーネル関数を起動する。実行結果を表５に示す。

表５　GPGPU での実行結果

	GTX1080	K40	C2050	k40host
視差画像全体転送(約200MB)	31.00	55.00	58.00	
(2)シフト加算	11.82	55.50	101.60	2973
(3)(4)評価と合成	0.53	1.41	2.04	37
最終結果転送(約1MB)	0.84	0.96	0.74	
計	44.19	112.87	162.38	3010.00

msec　　逐次

なお、表２の CPU での最速値、GPGPU にオフローディングするためのデータ転送時間も掲載した。シフト加算については 250 倍、評価と合成については 70 倍程度の高速化が得られている。本実験では、予め求めた視差画像の 200MB を転送するため、総計 44ms 程度の時間がかかっている。しかし、目標とするシステムでは、視差画像は別の手法で求める。たとえば、イメージセンサからの画像を転送し GPGPU などで視差画像を求める方式が考えられる。原画像の大きさは 50MB 程度なので、転送時間は短縮できる。また、転送とカーネル計算のオーバーラップを

CUDA で行うことで、転送時間を隠蔽するなどの工夫を考察することで、60fps の性能を確保できると考える。

6. マルチ GPGPU による高速化の検討

第 5 章で GPGPU によるライトフィールドカメラの処理について述べた。前章での高速化の結果は、640×480 ピクセルの解像度の結果である。さらなる高精細画像については、マルチ GPGPU の利用が考えられる。複数の GPGPU を用いる場合、次の二つの方法が考えられる。

(方式 1) マルチリフォーカス画像生成をマルチ GPGPU

図 6 の手順では、一つの GPGPU でマルチフォーカス画像 M=31 枚を生成していた。これをたとえば、8 枚×4 台とする方法である(図 16)。今後 d を増やしてフォーカスの精度を向上する場合に有効な方式である。

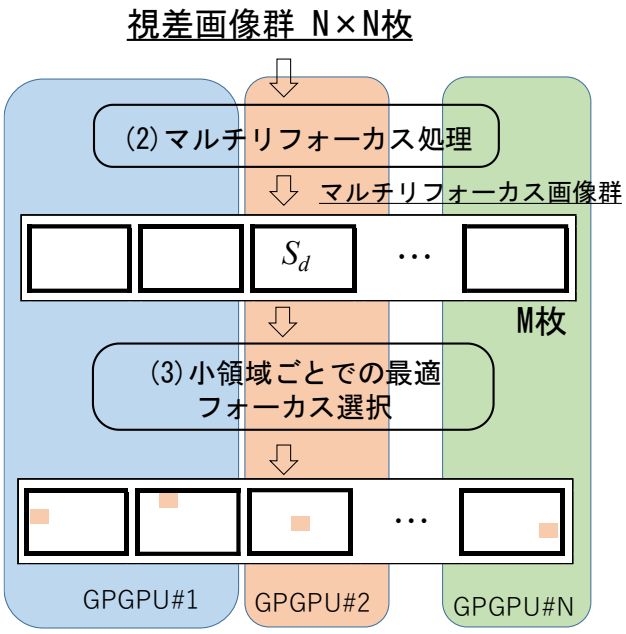

図 16　マルチ GPGPU によるリフォーカス画像生成

(方式 2) 部分領域ごとに各処理を適用

今回の画像の解像度は 640×480 ピクセルと K4、K8 に比べると小さいサイズである。4K、8K と解像度があがるにつれて、より計算時間は増大すると考えられる。そこで、画像を部分領域ごとに分割し、その分割した領域ごとに図 6 の(2)～(4)の処理を実行する。図 17 に GPGPU4 台での例を示す。

本理論では、画像の各領域で依存関係はない。端点が若干オーバーラップする程度だが、事前にオーバーラップ分のデータを視差画像群の部分領域に与えておけば良い。

原稿執筆時点で、全処理をマルチ GPGPU にした版は完成していないが、4 台の GTX1080 がマザーボード上の PCI-Express バスに挿入されたマシンで(2)のマルチリフォーカス処理について試験的に実行時間を確認した。結果を表 6 に示す。

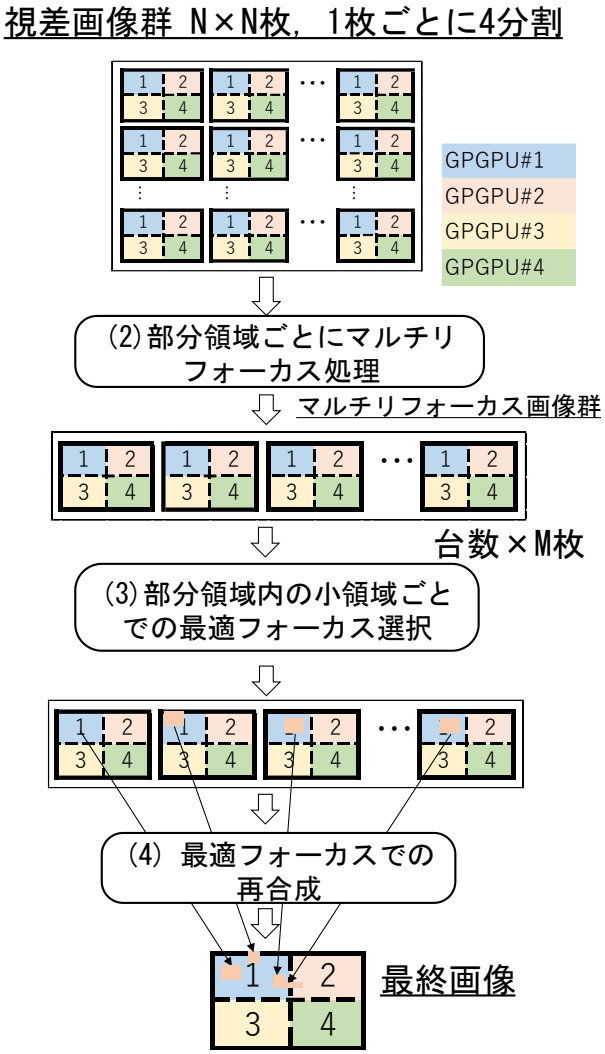

図 17 マルチ GPGPU による部分領域ごとの並列化

表 6　4 台の GTX1080 での実行時間

方式	プログラム番号	GPU逐次	ブロック	グリッド	メモリ配置	GTX1080 (msec)
1	M1	15x15	640, 1, 1	8, 480, 1	d-i-y-j-x	2.63
2	M2	15x15	320, 1, 1	31, 240, 1	d-i-y-j-x	2.66
2	M3	15x15	320, 1, 1	240, 31, 1	d-i-j-y-x	5.61
2	M4	15x15	320, 2, 1	120, 3, 1	d-i-j-y-x	5.60
2	M5	15x15	320, 3, 1	31, 80, 1	d-i-y-j-x	5.58

プログラム番号 M1 は方式 1 を用いた。31 枚の生成を 4 4 台の GPGPU で行った。 1 台あたり 8 枚を生成した。残りは、部分領域ごとに処理を行った。640×480 を 320×240 の画像を 4 枚と考えて実行した。いずれの場合も、最良値で見ると、1/4 の時間で実行できており、マルチ GPGPU の効果を期待できる。

この二つの方式については、シフト加算の結果では、ほぼ同じ性能となるが、視差画像の転送などを考えると、視差画像の大きさが 1/L (L は GPGPU の台数)になる方式 2 が有利ではないかと考えられる。特に、(3)(4)の後処理を考えると、画像の相互関係のない本方式では、中間結果を

相互転送する必要もないので、それぞれの GPGPU で独立に計算できる。マルチ GPGPU については、今後も検討を進める。

7. おわりに

本原稿では、ライトフィールドカメラの高速リフォーカス処理について述べた。通常の CPU による逐次実行では、640×480 の画像について、200 枚の視差画像からリフォーカス処理を行った場合 3 秒近い時間がかかっていたのが、12.3ms と 250 倍近い高速化が可能になった。また、複数台の GPGPU を用いるマルチ GPGPU 化についても検討し、4 台で 4 倍程度のシフト処理の可能性を示した。今後の課題は、性能解析ツールによる詳細な分析、マルチ GPGPU 化の実装、視差画像生成の検討があげられる。

謝辞 本研究はセコム科学技術振興財団「次世代画像入力システムを実現する高速パンチルト・リフォーカスカメラの開発」の支援を受けている。

参考文献

[1] R.Ng, M.Levoy, M.Bredif, G.Duval, M.Horowitz, P.Hanraha: Light field photography with a hand-held plenoptic camera, Stanford University Computer Science Tech Report CSTR 2005-02, 2005.
[2] B.Wilburn, N.Joshi, V.Vaish, E.Talvala, E.Antunez, A.Barth, A.Adams, M.Horowitz, M.Levoy: High performance imaging using large camera arrays, ACM Trans. Graphics, Vol.24, No.3, pp.767-776, 2005.
[3] 蚊野浩: コンピュテーショナルフォトグラフィーライトフィールドカメラ Lytro の動作原理とアルゴリズム，日本学術振興会，光エレクトロニクス第 130 委員会第 286 回研究会公開シンポジウム資料 2013.
[4] https://illum.lytro.com/
[5] J.C.Yang, M.Everett, C.Buehler, L.McMillan: A real-time distributed light field camera, EGRW '02 Proceedings of the 13th Eurographics workshop on Rendering, p.77-86, 2002.
[6] P.Alliez, K.Bala, K.Zhou: Real-time Depth of Field Rendering via Dynamic Light Field Generation and Filtering, Pacific Graphics 2010, Vol.29, No.7, 2010.