修士論文
「GPU を用いた画像処理の高速化」

2013 年 1 月 15 日

京都産業大学大学院
先端情報学研究科先端情報学専攻博士前期課程 2 年
鳴海 弘太郎

要約

本研究の目的は画像処理に GPU を利用する手法について検討し、空間フィルタやライトフィールドレンダリングなどのアプリケーションの動作を GPU の特徴である高い並列処理能力を利用する事により、高速化させる事である。本論文では、NVIDIA 社が提供する GPU 向けの統合開発環境 CUDA を使用した。まずは大規模な行列乗算を用いた実験によって、コアレッシング、タイル分割、繰り返し文の展開、プリフェッチ、粒度の調整といった GPU に適したプログラミング基本手法を提案し、それぞれの手法がどれだけ高速化されるかを比較した。次いで、それらの手法をベースとして、ラプラシアンフィルタ、動的計画法による画像中からのシームの検出、ライトフィールドレンダリングの高速化手法について研究した。その結果、256×256 画素のラプラシアンフィルタ処理で最大約 30 倍、256×256 画素の画像から動的計画法によってシームを検出する処理で約 90 倍、720×480 画素の 49 枚の画像を用いて 30 段階に焦点が異なる画像を生成するライトフィールドレンダリングで約 80 倍の処理速度の高速化を達成した。

内容

1.序論

GPU（Graphics Processing Unit）は、本来、3D グラフィックスの生成や表示に必要な処理に特化した専用プロセッサであったが、3D グラフィックに限らず、映像表示のコマ落ちや解像度の低下などを抑えて画面表示する処理などに利用された。これらにとどまらず、GPU が持つ高い並列処理能力にプログラム性を付加する事で、汎用的な計算に利用されるようになった。現在では、GPU 向けの開発環境が整ってきており、汎用的なコンピューティングに広く利用されている[1]。

画像処理は、膨大なデータを規則的に処理する場合が多く、計算コストが非常に高い。これまで、画像処理専用のハードウエアを開発することによって、この問題の解決が試みられてきた。しかし、画像圧縮・伸張やカメラ信号処理用のプロセッサなど一部の用途を除いて、画像処理プロセッサはそれほど利用されていない。これに対して、すでに広く普及している GPU による並列処理をさまざまな画像処理に導入することで、それらの多くを高速化できる可能が高い。実際、GPU を利用した画像処理の例として、動画像符号化の高速化[2]や局所不変特徴量の抽出[3]などの研究が行われている。

この研究では、GPU を利用した並列処理の高速化のための検討と、GPU を様々な画像処理に利用するための手法を提案する。第 2 章では GPU のアーキテクチャとそれを利用する開発環境である CUDA（Compute Unified Device Architecture）について説明した。第 3 章では行列演算を例として、コアレッシング、タイル分割、繰り返し文の展開、プリフェッチ、粒度の調整といった高速化の基本手法を検討した。第 4 章では、具体的な画像処理としてラプラシアンフィルタの実装、動的計画法による画像のリサイズ技術（シームカービング）、そしてコンピュテーショナルフォトグラフィ手法におけるライトフィールドレンダリングの実装に関して GPU での処理に適したプログラム手法の研究を行い、CPU（Central Processing Unit）と GPU の処理時間の比較を行った。なお、これらの開発には Visual Studio の C++と NVIDIA 社が開発・提供した GPU 向け統合開発環境 CUDA を用いた。

2.GPU と CUDA

この章では GPU と CUDA など、本論文で用いる重要な用語や概念を説明する。

2.1GPU

2.1.1GPU と CPU

GPU は、グラフィックボードなどに搭載され、3D グラフィックスの生成・表示に必要な計算処理を行うプロセッサである。同一の構造を持つ多数の要素プロセッサを内蔵しており、それらの要素プロセッサに処理を分散させ、演算を同時進行させる並列処理によって高速化を実現している。また、この高い並列処理能力を利用して、グラフィックス処理以外に活用する技術もある。そのように利用される GPU 技術を GPGPU(General Purpose Computing on GPU)と呼ぶ。

図 1 は CPU が行う逐次処理と GPU が行う並列処理の違いを概念的に示したものである。独立に処理できる 100 個の演算を逐次的に処理する場合、全ての演算が同じ時間で処理できると仮定すれば、処理時間の合計は演算 100 回分になる。一方、10 個の演算を完全に並列処理できる場合、処理時間は演算 10 回分になり、10 倍の高速化を達成できる。

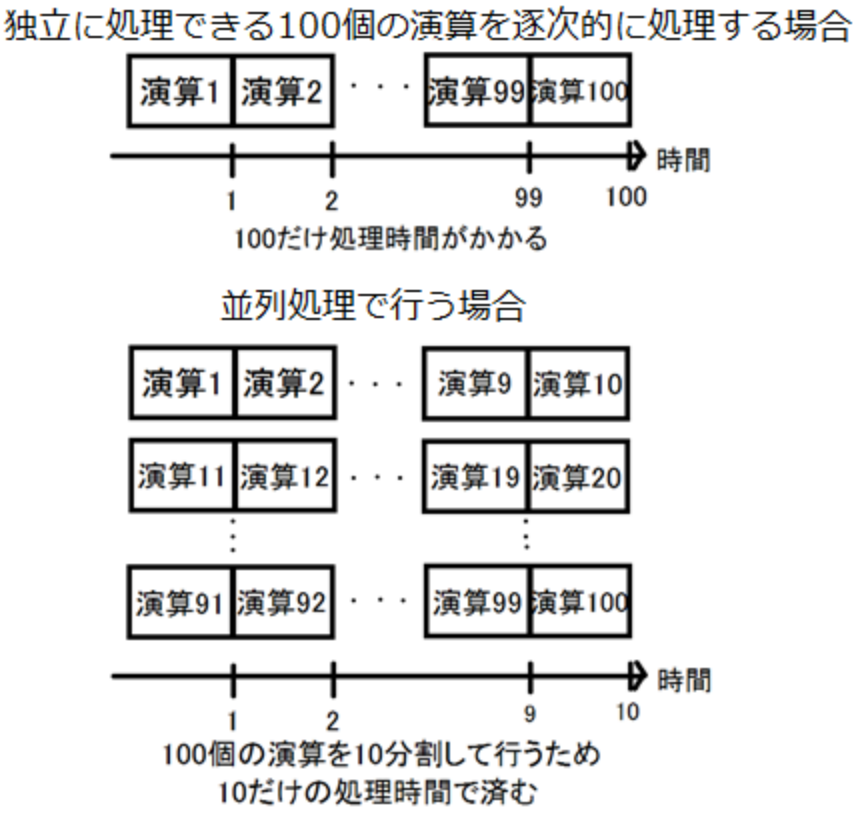


図 1 逐次処理と並列処理

CPU と GPU の、一般的な違いについて記述する。高性能な CPU は GPU よりもクロック周波数が高い。また、多くの命令を備え、様々な装置の制御ができるなどの汎用的な能力を備えている。しかし、スパコンなどの特殊な用途を除いて、プロセッサ数が少なく並列処理能力はそれほど高くない。CPU は、一昔前の世代から受け継いだシステムやデバイス（レガシー装置）を操作できる必要があり、メモリアクセスにも必要な制約が多い。

GPU は処理内容が限られるが、プロセッサ数が豊富である。また、メモリモデルが単純で制約が少ない GPU は、時間あたりに転送できる情報量(メモリ帯域幅)が大きいのが特徴である。今回使用する GPU である Tesla C 2075 のメモリ帯域幅は 144GB/s で、同じく今回使用する CPU の Intel® Xeon® Processor E5607 の 25.6GB/s に比べておよそ 5.6 倍の量である。ただし、前述した通り GPU が高速に処理できる事は限られているため、必要に応じて CPU と GPU を使い分ける事が総合的な処理能力の向上に繋がる。

2.1.2 GPU の構造

ここでは、NVIDIA 社の GPU ハードウエアの基本的な設計概念(アーキテクチャ)について説明する。NVIDIA 社の CUDA(後述)に対応した GPU のアーキテクチャは、概ね共通している。図 2 は CUDA に対応した NVIDIA 社製 GPU のアーキテクテクチャモデルである。マルチプロセッサ数は機種によって異なり、今回の研究に使った GPU である NVIDIA Tesla C2075 はマルチプロセッサ数が 448 基である。GPU は複数の「マルチプロセッサ」とメモリなどの周辺回路がブロック状に並んだ構成になっている。それぞれのマルチプロセッサごとに 8 個の「ストリームプロセッサ」が搭載されており、このストリームプロセッサによって並列処理が行われる。また、マルチプロセッサ内には、それぞれ「シェアードメモリ(共有メモリ)」とレジスタが搭載されている。これらのメモリは容量が小さくアクセスが高速である。そして、全マルチプロセッサからアクセスできる「ビデオメモリ(グローバルメモリ)」が存在する。このメモリは DRAM(Dynamic Random Access Memory)で実装されており、シェアードメモリなどに比べるとアクセス速度は遅く、容量が大きい。

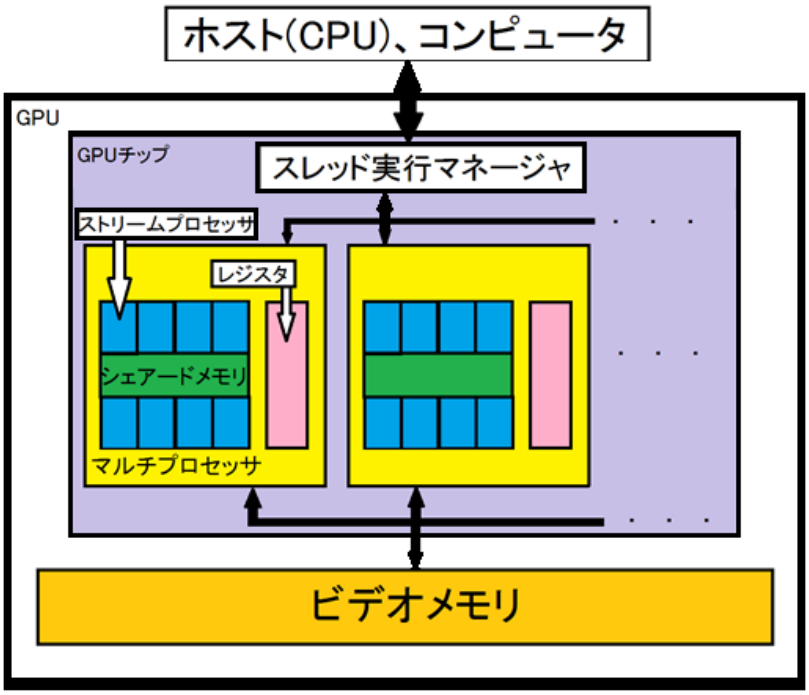


図 2 NVIDIA 社の CUDA 対応 GPU のアーキテクチャ

2.2CUDA

2.2.1CUDA とは

CUDA は、NVIDIA 社が開発した GPU での並列処理に特化した C 言語の統合開発環境であり、コンパイラ、およびライブラリなどから構成されている。Windows、Mac OS、Linux など様々な OS に対応している。また、Microsoft 社が提供する開発ツール Visual Studio(Visual C++)をサポートしており、設定を行う事によって Visual Studio からビルドする事が可能である。

NVIDIA のダウンロードサイト(http://developer.nvidia.com/cuda-downloads)から、ツールキットとサンプルコード、テンプレートを無償でダウンロードできる。今回の研究では、Windows 7 の Ver4.0 32bit を使用した。

2.2.2 CUDA における並列処理単位「スレッド」

GPU で並列処理を行うとき、その演算処理を並列化するための構造を設定する必要がある。CUDA では並列処理の単位を「スレッド」と呼ぶ。スレッドを 3 次元配列の構成にまとめたものを「ブロック」と呼ぶ。ソースコードにおいて GPU で実行する関数(カーネル関数)に 1 割り当てられるブロックを 2 次元配列の構成にまとめたものを「グリッド」と呼

ぶ。

CUDA で記述するプログラムの全体は C 言語ライクなものである。そのソースコードは CPU で実行される部分と GPU で実行される部分が混在している。カーネル関数は関数名に続いて<<<と>>>の間に 1 グリッド内のブロックの総数(スレッドブロック数)と 1 ブロック内のスレッドの総数(スレッドブロックのスレッド数)を表すパラメータを入力する。それらの数値が並列処理の度合いを決定する。1 つのブロックに 512 スレッドまで割り当てる事ができる。スレッドは GPU のストリームプロセッサに割り当てられ、処理される。

各ブロックに割り当てられたスレッドは、32 スレッドずつまとめられ、スケジューリングされる。これをワープと呼ぶ。ワープの要素はメモリアクセスなどで活用される。すなわち、あるワープがメモリアクセスを行いアクセス完了までに時間がかかってしまい、そのアクセスが完了するまでの遅延（レイテンシ）が発生するとき、ほかのワープがその間に処理を行う事によってレイテンシを補う事ができる。

グリッドに含まれる全てのスレッドは同じカーネル関数を実行する。そしてグリッド内のスレッド数の分だけ同時に並列処理を行う。同一グリッド内の全てのスレッドが同一のカーネル関数を実行する時、処理するデータの部分を区別するために、一意な座標を使用する。この座標を求めるために、あらかじめ定義された blockIdx、threadIdx、gridDim、blockDim という 4 つ構造体変数を使用する。blockIdx はスレッドが属するブロックの座標を保持し、threadIdx はブロック内のスレッドの座標を保持する。gridDim はスレッドが属するグリッドのブロックの構造を保持する。blockDim はブロック内のスレッドの構造を保持する。

例えば図 3 のようなブロック数 2×2、1 ブロックあたりのスレッド数 2×2×1 グリッド内のあるブロックの座標が(1,0)、そのブロックのあるスレッドの座標が(1,0)である場合、gridDim.x は 2、gridDim.y は 2、blockDim.x は 2、blockDim.y は 2、blockDim.z は 1、blockIdx.x は 1、blockIdx.y は 0、blockIdx.z は 0、threadIdx.x は 1、threadIdx.y は 0、threadIdx.z は 0 を示す。

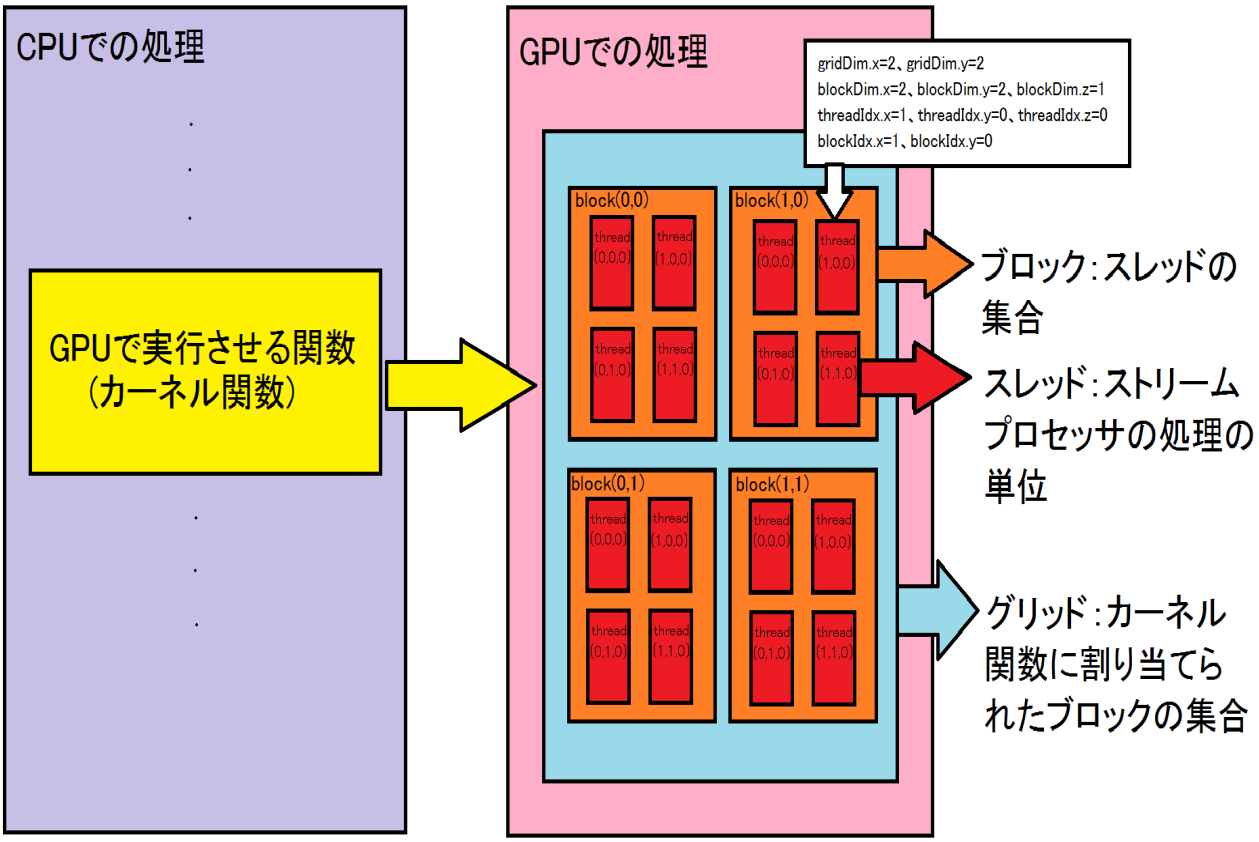


図 3 CUDA のアーキテクチャ

2.2.3CUDA のメモリモデル

図 4 のように、CUDA におけるスレッドはレジスタとローカルメモリを有する。そして、同じブロック内のスレッドが共有するシェアードメモリがある。さらに同一グリッド内にはグローバルメモリ、コンスタントメモリ、テクスチャメモリがある。

レジスタは特定の用途に利用されるごく小容量の高速メモリである。ローカルメモリは GPU チップ外のメモリデバイスに配置されたオフチップメモリである。主にローカル変数の退避領域として利用され、アクセス速度は低速である。シェアードメモリは容量が小さくアクセス速度が速い。グローバルメモリはシェアードメモリよりも速度が遅いが、容量が大きい。コンスタントメモリとテクスチャメモリはグローバルメモリに比べてアクセス速度が速いが、CPU 側からはライトオンリー、GPU 側からはリードオンリーである。表 1 に今回の実験で使用する Tesla C2075 の物理的なメモリと CUDA のメモリモデルの対応関係を示す。

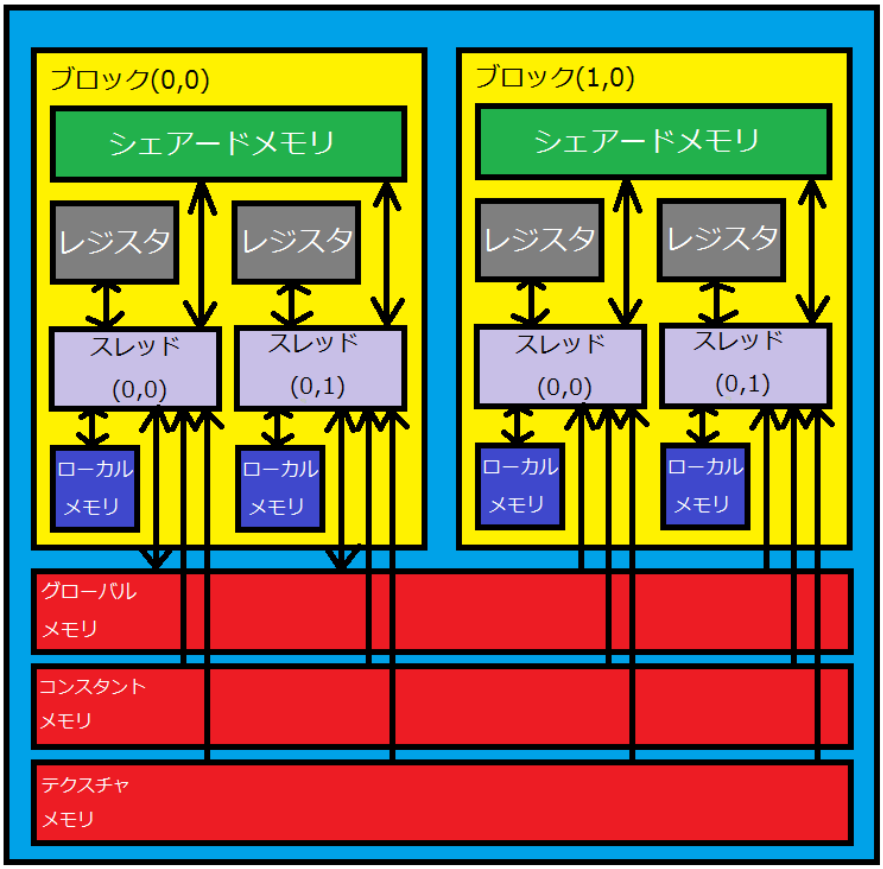


図 4 CUDA のメモリモデル

表 1 Tesla C2075 のメモリごとのスペック

CUDA のメモリ	対応する GPU の物理メモリ	容量	速度	GPU からの読み書きの可否
レジスタ	レジスタ	786KB	非常に高速	共に可能
シェアードメモリ	シェアードメモリ	49KB	非常に高速	共に可能
グローバルメモリ	ビデオメモリ	4096MB	低速	共に可能
コンスタントメモリ	ビデオメモリ	65KB	高速	読み出しのみ可能
テクスチャメモリ	ビデオメモリ	512MB	高速	読み出しのみ可能
ローカルメモリ	ビデオメモリ	2147MB	低速	共に可能

2.2.4 ソースコードの記述

```
1
2  #include <stdio.h>
3  #include <cutil.h> //CUDA用のヘッダ
4
5  #define N 100 //メモリの大きさ用の数字
6
7  __global__ Kernel1(float *d_a){
8
9  //...
10
11 }//GPUでの処理のコード
12
13 int main(int argc, char* argv){ //メイン関数
14     //...
15     CUT_DEVICE_INIT(argc,argv); //GPUの初期化
16     //...
17     float *d_a, h_a;//CPU,GPUそれぞれのメモリ、ポインタを宣言
18     //...
19     cudaMalloc((void**)&d_a, sizeof(float)*N); //GPUにd_aというメモリ領域を100だけ確保
20     //...
21     cudaMemcpy(d_a, h_a, sizeof(float)*N, cudaMemcpyHostToDevice);//メモリh_aのデータを丸々d_aにコピー
22
23     dim3 block(a,b,c);  //1グリッドあたりのブロック数(a×b×c個)と、
24     dim3 thread(d,e,f); //1ブロックあたりのスレッド数(d×e×f個)の宣言
25
26     Kernel1<<<block, thread>>>(d_a)//ここで、GPU内で処理する関数の実行
27
28     cudaMemcpy(h_a, d_a, sizeof(float)*N, cudaMemcpyDeviceToHost);//処理されたメモリd_aのデータを丸々h_aにコピー
29     //...
30     cudaFree(d_a); //メモリd_aを破棄
31
32     CUT_EXIT(argc, argv); //終了
33
34     return 0;
35
36 }
37
```

図 5 CUDA のソースコードのサンプル

図 5 に CUDA によるプログラムのソースコードの例を示す。ファイルの拡張子は.cu である。基本的な構文は C 言語と同じだが、3 行目にある CUDA 用のヘッダファイルである cutil.h をインクルードさせることで CUDA に特有なプログラミングが可能になる。7～11 行目は GPU で処理を行う関数である。 __global__を先頭につける事で GPU 処理が可能となっている。メイン関数は基本的に CPU で処理を行うが、GPU で並列処理を行うために、ここで以下の手順を踏む必要がある。

・GPUデバイスを初期化する(CUT_DEVICE_INIT)(15行目)
・GPUデバイス上にメモリを確保する(cudaMalloc)(19行目)
・CPU側で演算するためのデータを生成し、CPUからGPUのメモリへデータを送る(cudaMemcpy)(21行目)
・スレッド数とブロック数を定義する(dim3)(23,24行目)
・GPU側で行う動作(カーネル)を実行する(26行目)
・処理結果のデータをGPUからCPUへ送る(cudaMemcpy)(28行目)
・GPUのメモリを解放する(cudaFree) (30行目)

19 行目に書かれた cudaMalloc でメモリを確保する場合、cudaMalloc 内の第 2 引数の部分に sizeof((型)*(数))を入力することで、確保するメモリ容量を設定する(図 5 では、float 型の領域 100 個分を確保した)。また、cudaMemcpy でメモリ内のデータをコピーする場合、デバイス(GPU)とホスト(CPU)のどちらからどちらへコピーするかを表すために、21 行目のように cudaMemcpyHostToDevice(ホスト側からデバイス側へ)、28 行目のように

cudaMemcpyDeviceToHost(デバイス側からホスト側へ)といった記号定数が必要になる。

3.GPU 処理の最適化

前章で述べたように、CUDA は汎用的な並列処理のプログラミングが可能である。 しかし、単純に並列化を行っただけでは、容易にメモリ帯域幅の上限に達してしまい、並列処理の恩恵を受けることができなくなってしまう。これを防ぐためには、グローバルメモリなどへのメモリアクセスを極力減らし、メモリアクセスに対する演算の比率を大きくするなどの工夫が必要である。

この章では、大規模な行列の積という演算を例にして、並列処理を最適化する手法を挙げ、処理速度が改善される程度を比較する。なお、開発環境は表 2 のものを利用した。

表 2 本実験の開発環境

開発環境	
OS	Windows 7 Professional 64bit
PC	HP Z800 Workstation
CPU	Intel® Xeon ® CPU E5607 2.27GHz
GPU	NVIDIA Tesla C2075 1.15GHz
開発ツール	Microsoft Visual Studio 2008
CUDA バージョン	4.0
グローバルメモリ 容量	4096MB
シェアードメモリ 容量	49KB
ストリーミングプロセッサ の数	448 基

3.1 GPU によって高速化が可能なアルゴリズムの原則

多くの画像処理は、完全並列化が可能なように思われるが、単純に並列化しただけでは処理の大幅な高速化は望めない。その理由の多くは、グローバルメモリへのアクセスに処理時間の大半を要してしまい、ほとんどの要素プロセッサがメモリアクセス待ちになってしまうことである。グローバルメモリにアクセスする場合、1 回ごとに長いアクセス時間を要し、その結果 GPU の処理が待たされるというレイテンシが発生する。処理時間に占めるレイテンシの割合を小さくするためには、グローバルメモリへの 1 回の読み込みのアクセスに対してそのアクセスしたデータを利用した演算の回数の割合を大きくすることが必要である。この割合の数値を CGMA(Compute to Global Memory Access)率という。CGMA 率は GPU を利用した並列処理において、高速化できる割合の目安となる。

たとえば、グローバルメモリにある画像データを使用し、出力画素値= 注目画素値-注目画素の右隣の画素値という微分画像を生成する処理では、２つの画素値の読み込みが行われるので、合計 2 回のメモリアクセスが行われる。このアクセスしたデータを使用した浮動小数演算は 1 回の減算のみである。この時、浮動小数演算回数÷メモリアクセス回数は 0.5 となる。これではグローバルメモリへのアクセスに対する演算の数が少なく、GPU を導入しても高速化の恩恵は少ない。

より高速化させるためには、GPU 内に存在するシェアードメモリやレジスタなどの高速にアクセスできるメモリを利用してそれらを経由させる。それによってグローバルメモリへのアクセスの比重を減らし、CGMA 率を上昇させることが可能である。

3.2 最適化の手法

3.2.1 コアレッシング

C 言語及び、CUDA で行列要素を構成する場合、配列要素はメモリ上で行優先に並んでいる。つまり、N×N 行列のとき、行 0 の N 個の全ての要素が順に並び、次いで、行 1 の要素が並ぶ。図 6 は 4×4 行列の配置を図示したものである。この配列に複数のスレッドが同時にメモリアクセスするとき、アクセスする要素がメモリ上で連続したアドレスに配置されておれば、DRAM の高速アクセスモードを利用することができる。これによって、メモリへのアクセス時間を大幅に減らす方法をコアレスアクセスと呼び、コアレスアクセスを行わせる事をコアレッシングと呼ぶ。

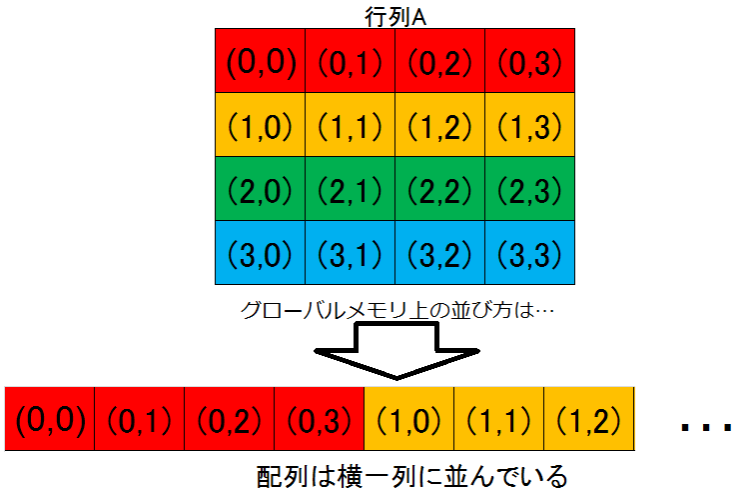


図 6 メモリ上の配列要素の並び

図 7 と図 8 は、4×4 の行列の要素を 4 つのスレッドに 4 要素ずつアクセスさせる処理を図示したものである。

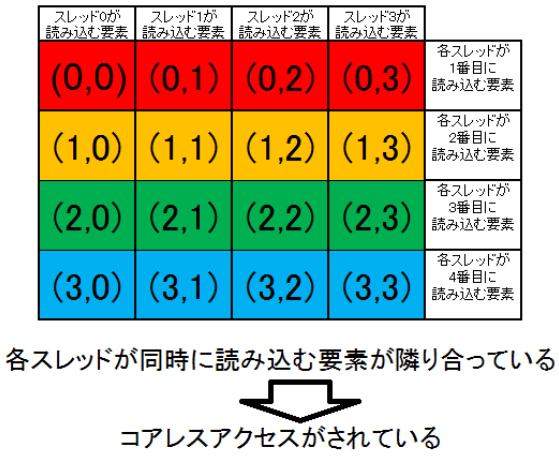


図 7 コアレスアクセスしているアクセス方法

図 7 はそれぞれのスレッドが列方向にアクセスし、そのような複数のスレッドが行方向に同時にメモリアクセスする場合を図示したものである。複数のスレッドが配列のメモリにアクセスする際、それらは同時に行 0 にアクセスしている。行 0 の要素はメモリ上で隣り合っており、このときにコアレスアクセスが行われ、アクセス時間が短縮される。

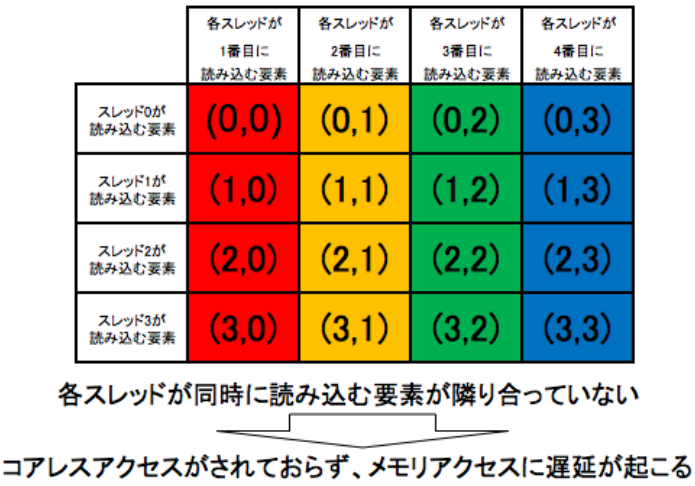


図 8 コアレスアクセスしていないアクセス方法

図 8 はそれぞれのスレッドは行方向にアクセスし、そのような複数のスレッドが列方向に同時にアクセスする場合を図示したものである。複数のスレッドが配列のメモリにアクセスする際、それらは同時に列 0 にアクセスしている。列 0 の要素はメモリ上で隣り合っておらず、このときはコアレスアクセスが行われず、アクセスに遅延がおこってしまう。

3.2.2 タイル分割

グローバルメモリはレイテンシが大きくランダムアクセスに弱いという特徴を持っている。グローバルメモリ上の特定の不連続データを複数回利用する場合、要素をシェアードメモリにコピーしてから使用する事で、アクセス速度を改善する事ができる。

今回の様な行列乗算を行う時、計算に使う行列を CUDA の処理単位の一つであるブロックに分割させて乗算を行う。この時、各ブロック内ではそのブロックで利用する全ての数値が複数回使われる。そのため、この数値を毎回グローバルメモリから読み出すと、それに応じたレイテンシが生じる。そこで、ブロック内の要素をシェアードメモリに移し、その数値を利用して演算を行う事により、グローバルメモリへのアクセス回数を減らす。その結果、処理を高速化させる事が可能となる。このように配列の要素を部分的に分割し、それぞれをシェアードメモリに転送した後、演算させる処理をタイル分割という。

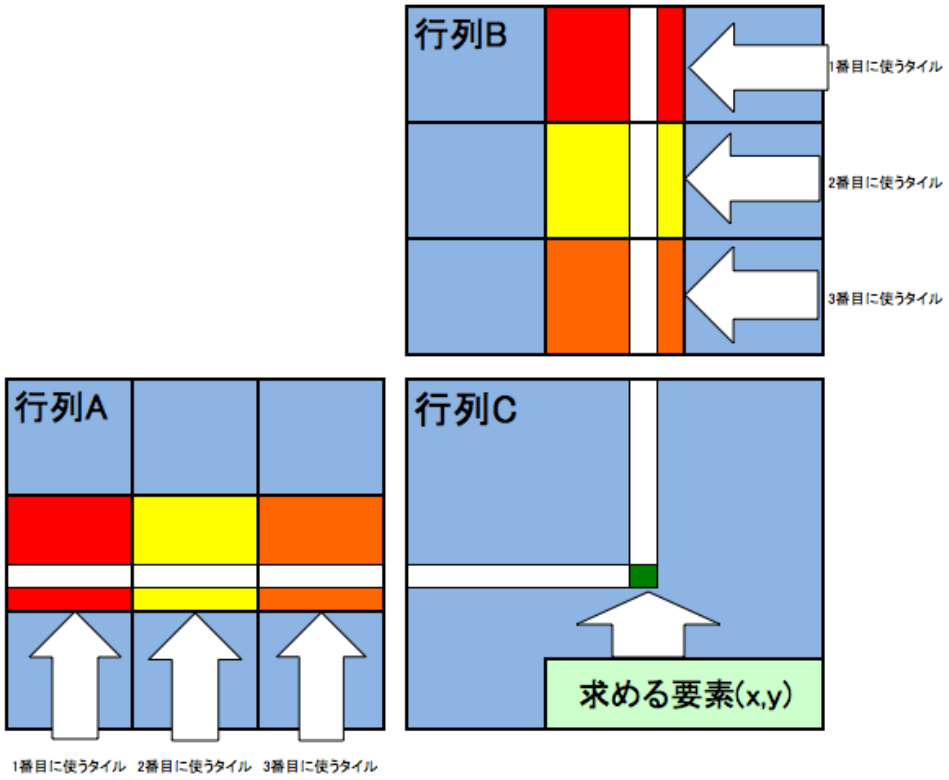


図 9 タイル分割した行列積算

図 9 は行列 A と行列 B を乗算して行列 C を求める時のタイル分割処理を図示したものである。流れとしては、グローバルメモリに格納された行列 A と行列 B の全係数から、1 ブロック分の部分行列(タイル)をシェアードメモリに書き込む。続いて書き込んだシェアードメモリを用いて部分行列の乗算を行う。乗算が完了したら次のタイルをシェアードメモリに書き込み乗算を行う。これを繰り返す事によって最終結果を高速に計算することが可能である。

3.2.3 命令の展開

GPU における各要素プロセッサの処理速度には限界がある。分岐、繰り返しの処理も、メモリアクセス同様に GPU の処理時間を消費する。高い頻度で繰り返し命令を利用すると、それらの浮動小数点命令に対する割合を増やしてしまい、全体として処理時間が大きくなってしまう。たとえば図 10(a)のようにループカウンタを使用すると、浮動小数点計算の命令を 2 つ、命令分岐を 1 つ、アドレス計算の命令を 2 つ、ループカウンタ加算の命令を 1 つ実行する。このとき、浮動小数点計算の命令は全体の 1/3 となる。

図 10(b)は行列 As と行列 Bs の行列乗算を行う繰り返し文を展開したソースコードである。これによって分岐命令とループカウンタ加算を省く事ができ、処理時間のうちの浮動小数点計算処理の割合が増える。この場合、浮動小数点計算の命令を 32 回、アドレス計算を 1 回実行することになる。浮動小数点計算の割合が 31/32 となるので、処理の殆どの部分を

浮動小数点計算に使うことができ、処理時間を短縮できる。

```
for (int k = 0; k < BLOCK; k++) {
    tmp += As[ty][k] * Bs[k][tx];
}
```

a.forループで括った場合

```
tmp +=  As[ty][0] * Bs[0][tx] + As[ty][1] * Bs[1][tx] + As[ty][2] * Bs[2][tx] + As[ty][3] * Bs[3][tx]
    + As[ty][4] * Bs[4][tx] + As[ty][5] * Bs[5][tx] + As[ty][6] * Bs[6][tx] + As[ty][7] * Bs[7][tx]
    + As[ty][8] * Bs[8][tx] + As[ty][9] * Bs[9][tx] + As[ty][10] * Bs[10][tx] + As[ty][11] * Bs[11][tx]
    + As[ty][12] * Bs[12][tx] + As[ty][13] * Bs[13][tx] + As[ty][14] * Bs[14][tx] + As[ty][15] * Bs[15][tx];
```

b.ループ文の展開

図 10 ループ文の展開

3.2.4 プリフェッチ

メモリアクセスを行う場合一度にアクセスできる量の制限が存在する(メモリ帯域幅)。GPU とグローバルメモリの間には、複数のアクセスパスがある。そのためメモリ帯域幅が CPU よりも大きい。グローバルメモリにアクセスする時、アクセスパスが DRAM であるためアクセス速度は遅く、複数のプロセッサが同時にアクセスを要求したとき、メモリ帯域幅を簡単に超えてしまいアクセスが完了するまでの遅延(レイテンシ)だけでなく、プロセッサがアクセスするためにアクセスパスを使用するための順番待ちが生じてしまう。タイル分割を使用し、シェアードメモリを経由して処理を行うことで帯域幅の制限に対処できる。また、CUDA のスレッドでは、ワープがメモリアクセスを待つ間にほかのワープを進める事によってメモリアクセスによるレイテンシを隠蔽して処理を行う。しかし、グローバルメモリからシェアードメモリに移したデータをすぐに演算に使ってしまうなど、メモリアクセスと演算の間に入る処理が少ない場合、使用するメモリの切り替えのために長いレイテンシが生じてしまう。しかもこの場合、全てのスレッドがメモリアクセスを行っている状態になりやすくなり、これではレイテンシが隠蔽しきれないため、結果的に処理時間が長くなってしまう。

この問題を解決する方法は、現在のデータ要素の処理中にメモリアクセスとアクセスされたデータの処理の間に独立した処理を行わせることである。これによって、スレッドがグローバルメモリのアクセスするデータを待たなければならない時間を低減させることができる。

前述したタイル分割の処理を行う際、現在のタイルの処理中に、次に処理を行うタイル要素をレジスタに読み込ませる。これによって、1 回のメモリアクセスとそのアクセスされたデータに対する処理の間に、独立した処理の量を挿入することで帯域幅制限と長い各スレッドに起こるレイテンシを隠蔽することができる。これをプリフェッチと呼ぶ。

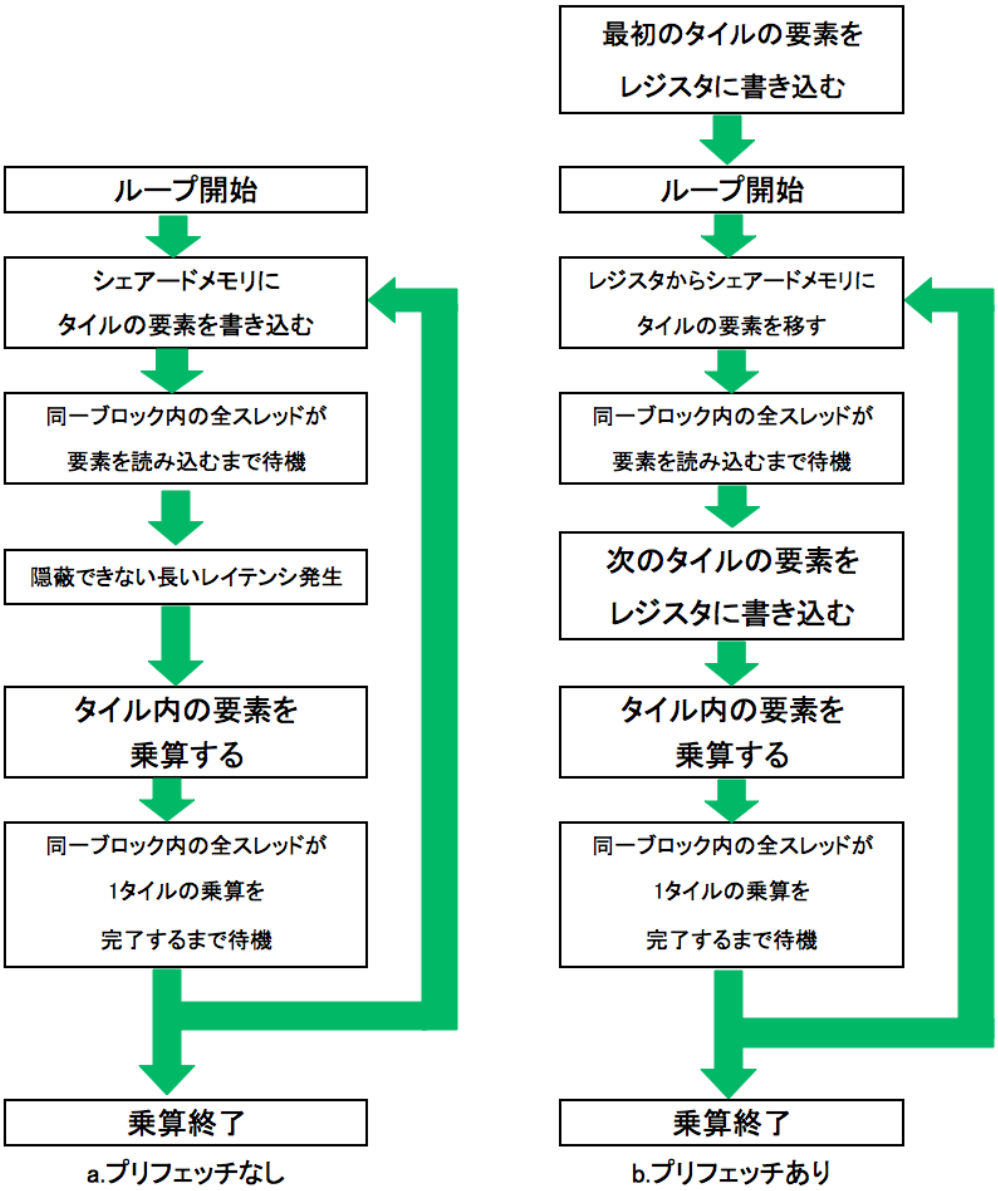


図 11 プリフェッチの流れ

図 11 はプリフェッチの有無による処理の流れを比較したものである。 まず(b)では、全スレッドが最初のタイル要素をグローバルメモリからレジスタに読み込む。そして(a)、(b)ともにタイルを使用して行列乗算を行う処理を開始する。このとき、(a)のグローバルメモリからシェアードメモリへタイルを書き込む時間と、(b)で次のタイルの要素をグローバルメモリからレジスタに書き込む時間にそれほど差はないが、(a)はタイルを読み込んだあとすぐにそのアクセスしたデータで演算を行うため、グローバルメモリにアクセスするためのレイテンシが生じてしまう上に、それを隠蔽できなくなる。一方(b)ではこのレイテンシの間にレジスタからシェアードメモリにタイルの要素を移していく。この 2 つのメモリはともに非常に高速にメモリアクセスすることができ、アクセス時間は(a)のレイテンシより短い。この処理により、レイテンシは隠蔽されるが、先読みする要素の分だけレジスタを

多く使用してしまうため、メモリが飽和してしまう恐れがある。

3.2.5 スレッド粒度の調整

図 12 は、一度に複数のタイルを読み込み、出力する要素を 1 スレッドにつき複数求めるようにする処理を図示したものである。各スレッドにより多くの処理を任せ、処理に必要なスレッドを減らす。これにより、各スレッド間の連絡の遅延「並列スローダウン」を軽減するなどのメリットがある。デメリットとして、大量のメモリを使用するため、並列性が不十分になる恐れがある。

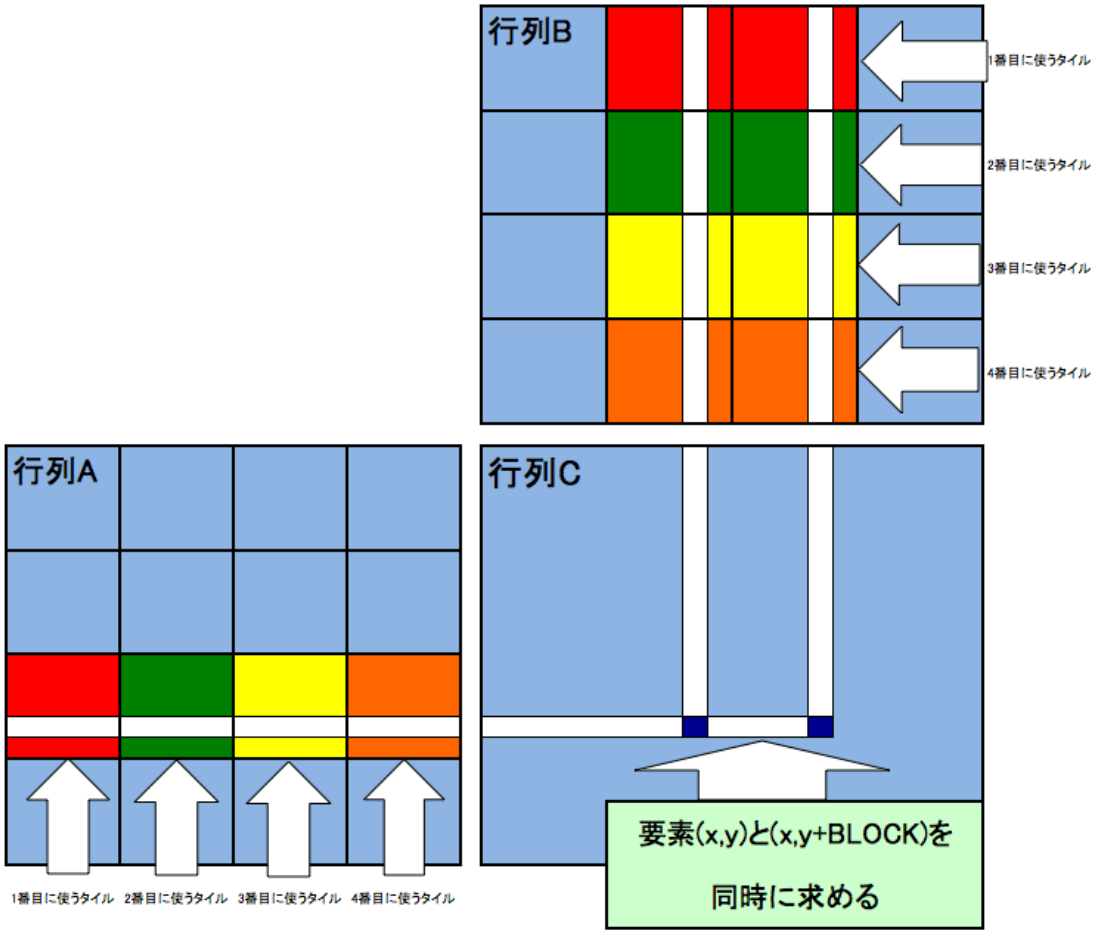


図 12 1 つのスレッドで複数の要素を求める処理

3.3 手法による処理時間の比較と考察

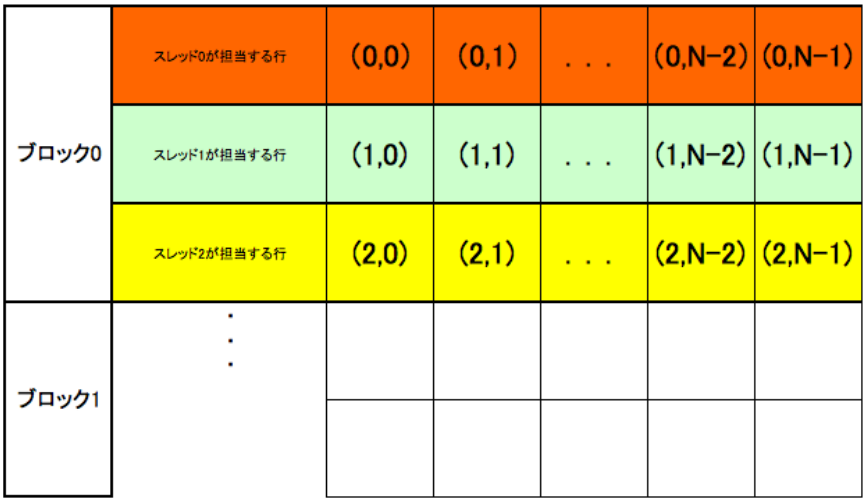
これらの工夫を行列乗算のプログラムに組込むことで、処理がどの程度高速化されるかを検証する。今回の実験は、512×512 の行列で、その係数が同じ 2 つの浮動小数点の行列を乗算させ、最後の要素の積和が終了するまでの時間を計測した。計測時間から 1 秒あたりの浮動小数点計算量(FLOPS)を導きだし、グラフで表した。 今回の実験での FLOPS は N×N×N×2÷処理時間(N は行列の 1 辺の長さ)で求めた。グラフの縦軸の値は全て FLOPS で表し、それぞれの処理をスレッド数が 4×4、8×8、16×16 で行う。なお、3.3.2 以降の結果は全てコアレッシングが行われている。また、CPU によって逐次処理を行わせた結果は 43,738,756FLOPS（43.74MFLOPS）となった。

3.3.1 コアレッシング

コアレスアクセスの有無による処理時間の比較を行う。図 13 は処理方法を図示したものである。 (a)のコアレッングを行わない処理では、行ごとにスレッドを対応させ、一つのスレッドが 1 行分の要素を乗算する。 一方(b)のコアレッシングを行う処理は、列ごとにスレッドを対応させ、一つのスレッドが 1 列分の要素を乗算する。

表 3 と図 14 にコアレッシングの有無による処理速度の比較を示す。これらから明らかなように、処理速度に約 10～80 倍の違いがあらわれた。コアレッシングが行われている場合では、スレッド数が増えるごとに GPU 内で使われるストリームプロセッサが増えるため、処理速度が増えていく。逆に、コアレッシングが行われていない場合はスレッドが増えていくにつれ、処理速度が低下していった。これは、スレッド数の増加に伴い、グローバルメモリに同時にアクセスする回数が増えていき、アクセスが衝突することが原因であると考えられる。アクセスが衝突した場合、プロセッサが一度にアクセスできるメモリの量をすぐに上回ってしまい、一回のアクセスでは必要な量のメモリアクセスができないため、すべてのメモリアクセスを行うためにアクセス処理を複数回反復で行うことになる。このため遅延が起こってしまう。この遅延はスレッドが増えるにつれてどんどん大きくなっていきスレッド数が 16×16 のときには CPU のときと同じ程度の処理速度となってしまう。

以上のことから、GPU で並列処理を行う上ではコアレッシングを行うことは不可欠であるといえる。



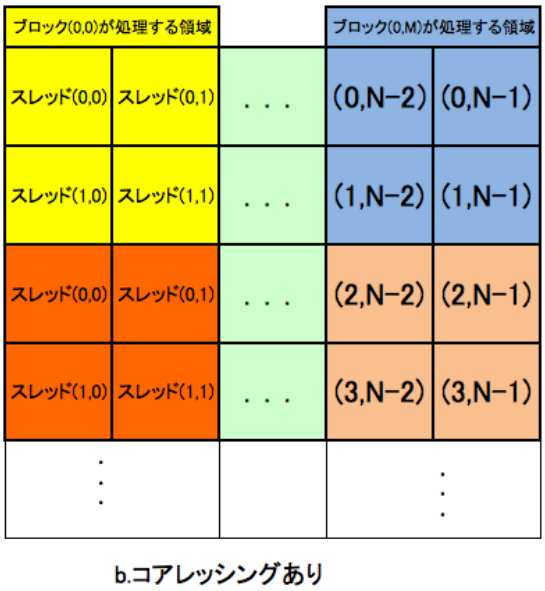


図 13 コアレッシングの処理方法

表 3 コアレッシングの有無による処理速度の比較[MFLOPS]

スレッド数	コアレッシングあり	コアレッシングなし
4×4	872.28	95.08
8×8	2339.48	51.93
16×16	3187.93	41.11

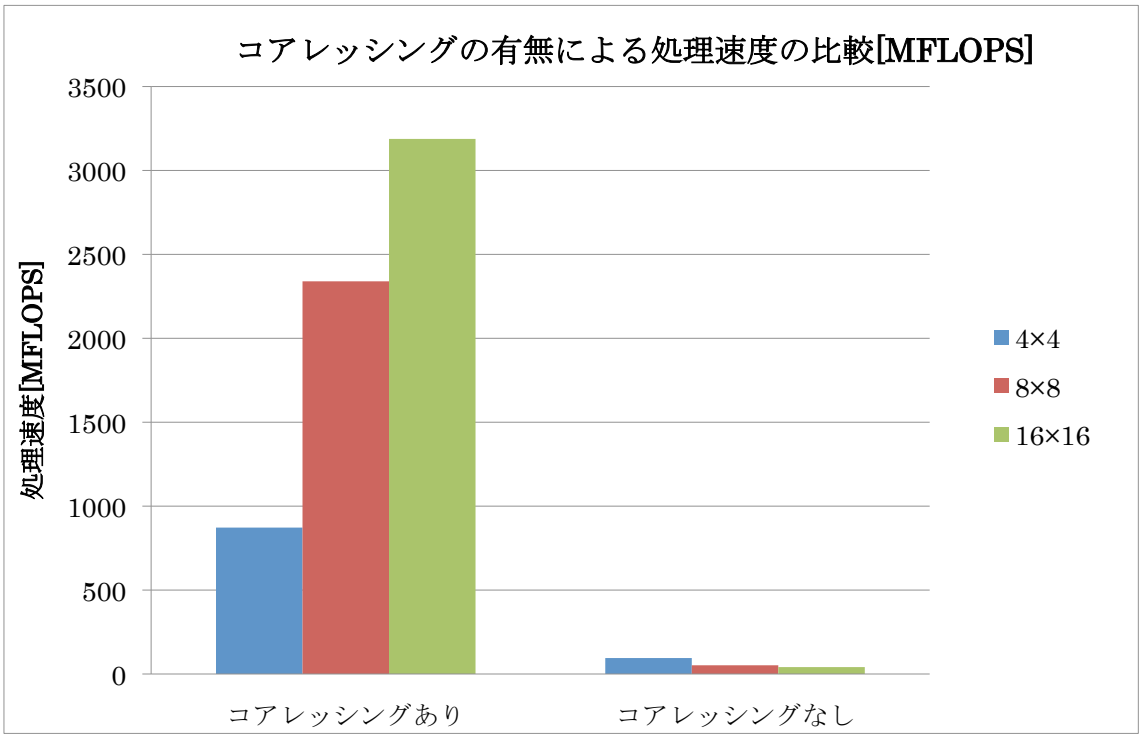


図 14 コアレッシングの有無による処理速度の比較

3.3.2 タイル分割

タイル分割によるシェアードメモリを使用した処理と、3.3.1 で行ったグローバルメモリのみを使用した処理の処理速度を比較する。タイル分割による処理の場合は、答えの行列をブロック分だけ小領域に分割し、1 スレッドにつき 1 要素計算する。さらに、その小領域(タイル)内で共有するシェアードメモリに各ブロックごとに積算する 2 つの部分行列の要素を移し、それらを用いて部分行列の積算を行う。

表 4 および図 15 からわかるように、シェアードメモリを利用して行列乗算を行った結果、グローバルメモリのみを使用するときに比べ、最大 8 倍以上の高速化に成功した。グローバルメモリのみを使用する場合 CGMA 率は 1.0 だが、シェアードメモリを使用してタイル分割を行う場合は、各スレッドがタイルのかけられる行列の 1 行分とかける行列の 1 列分のシェアードメモリの要素にアクセスをするようになるため、CGMA 率は定義したタイルの大きさ(定義したスレッドの 1 辺分)だけ倍増し、非常に高速な処理が可能になる。このことから、グローバルメモリの同一要素に複数回のアクセスが発生する場合、その内容をシェアードメモリに移動させ、シェアードメモリにアクセスすることが有効であることを確認できた。

しかし、グローバルメモリのみを使用した場合に比べて、タイルを利用した場合はスレッドを増やすことの効果は小さかった。グローバルメモリのみを利用する場合はスレッドが増えればその分使用するストリームプロセッサも増え、ある程度高速化された。しかし、シェアードメモリからアクセスする場合、GPU では 16 スレッドずつ(16 スレッドの集まりをバンクという)。同じバンクに属するスレッドがシェアードメモリ内の同じ場所に同時にアクセスを要求した場合、1 スレッドずつ順番にアクセスされる。順番に処理しなければならない分処理時間がかかってしまい、これをバンク・コンフリクトという。スレッドが増えれば増えるほどシェアードメモリ内の 1 つの要素にアクセスするスレッドの数も増えるため、スレッドが増えてもそこまで処理速度が増えない。このバンク・コンフリクトは、このあとのシェアードメモリを使用した実験でも起こる。

表 4 タイル分割の有無による処理速度の比較[MFLOPS]

スレッド数	タイル分割	グローバルメモリのみ
4×4	6,659.77	872.28
8×8	6,925.76	2339.48
16×16	7,502.81	3187.93

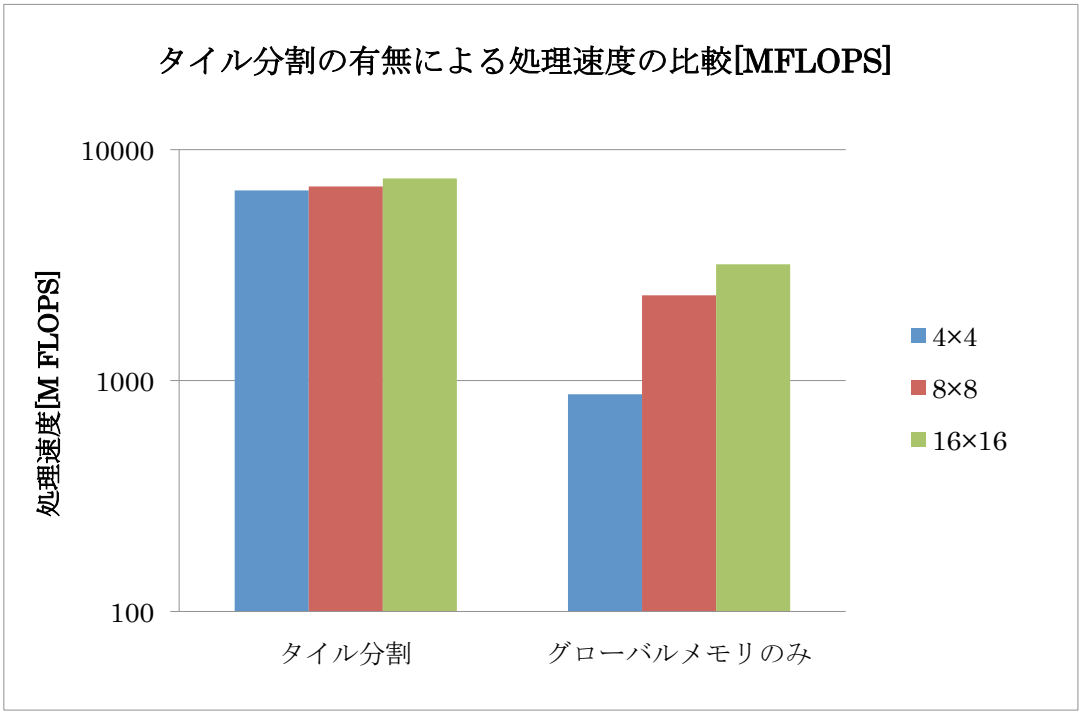


図 15 タイル分割の有無による処理速度の比較

3.3.3 ループの展開

タイル分割の手法に加えて、実際の要素を求める演算で繰り返し文を使わず、演算に使用する要素を直接指定する。ループを展開させる前の処理速度と展開させた後の処理速度、およびグローバルメモリのみを使用した場合の処理速度を比較する。

表 5 および図 16 からわかるように、ループ展開を行うことで、約 1.3 倍の高速化に成功した。ループを展開することによって、ループカウンタの加算などの演算結果に関わらない処理を減らし、処理時間を短縮することができた。

表 5 ループ展開を行った際の処理速度の比較[MFLOPS]

スレッド数	タイル分割	タイル +ループ展開
4×4	6,659.77	9,118.67
8×8	6,925.76	9,878.76
16×16	7,502.81	10,221.05

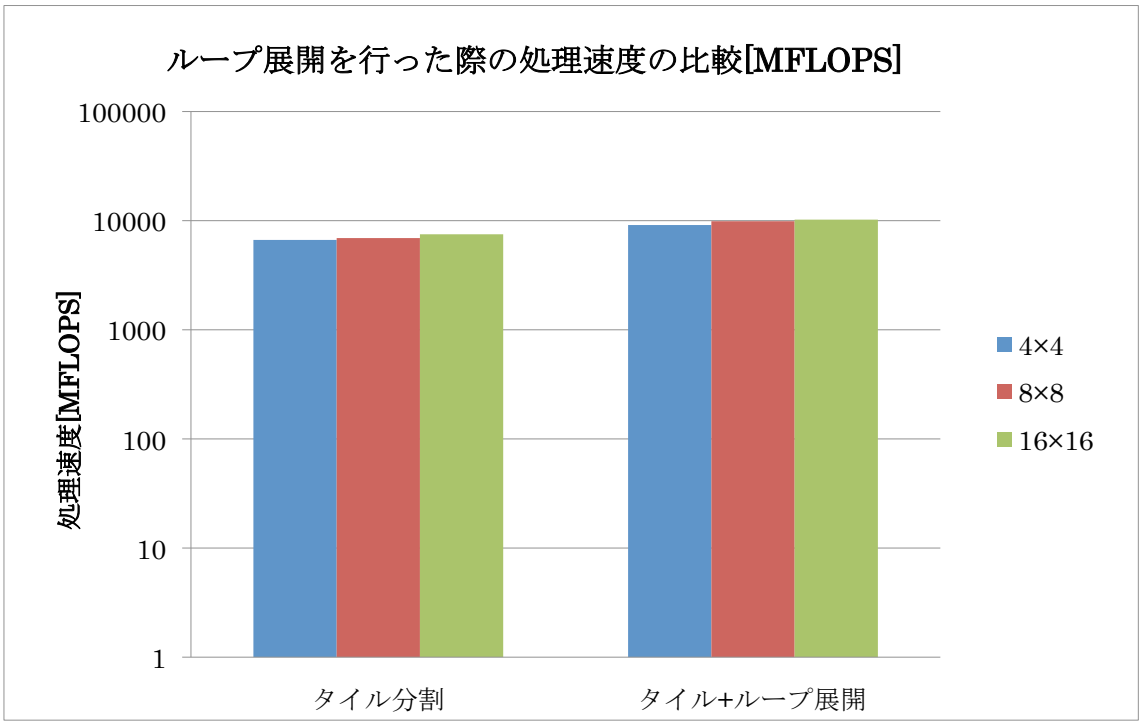


図 16 ループ展開を行った場合の処理速度の比較

3.3.4 プリフェッチ

これまでの処理に加えて、それぞれのスレッドで要素を演算する前に、次の演算に使用するタイルをあらかじめレジスタに送り、次の処理でレジスタから要素を読み込ませるという処理を繰り返させる。今回はこのプリフェッチを行った処理と行っていない処理をそれぞれループ展開しているときとしていないときで比較した。

表 6 および図 17 が示すように、プリフェッチを行った処理は、プリフェッチを行わない処理に比べて最大約 450MFLOPS の高速化に成功した。しかし、ループ展開に比べて高速化の度合いは低かった。レイテンシの間に、繰り返し処理の中の次の処理のためのメモリアクセスを行ったため、メモリアクセス自体は増えていることが原因と考えられる。しかし、メモリアクセスの速度が速いレジスタを経由してタイルが生成されるため、処理速度は単純なタイル分割よりも向上した。ただし、1 ブロックあたりのスレッド数を増やして 1 つのタイルを大きくするにつれて、レジスタへのアクセスの量が増えすぎて処理速度がかえって下がってしまうという事もあった。このことから、プリフェッチを行う際は、1 ブロックあたりのスレッド数をより考慮して行う事が重要である。

表 6 プリフェッチを行った際の処理速度の比較[MFLOPS]

スレッド数	タイル分割	タイル +プリフェッチ	タイル +ループ展開	タイル+プリフェッチ +ループ展開
4×4	6,659.77	6,812.91	9,118.67	9560.01
8×8	6,925.76	7,599.02	9,878.76	9,878.76
16×16	7,502.81	7,876.86	10,021.05	9,716.77

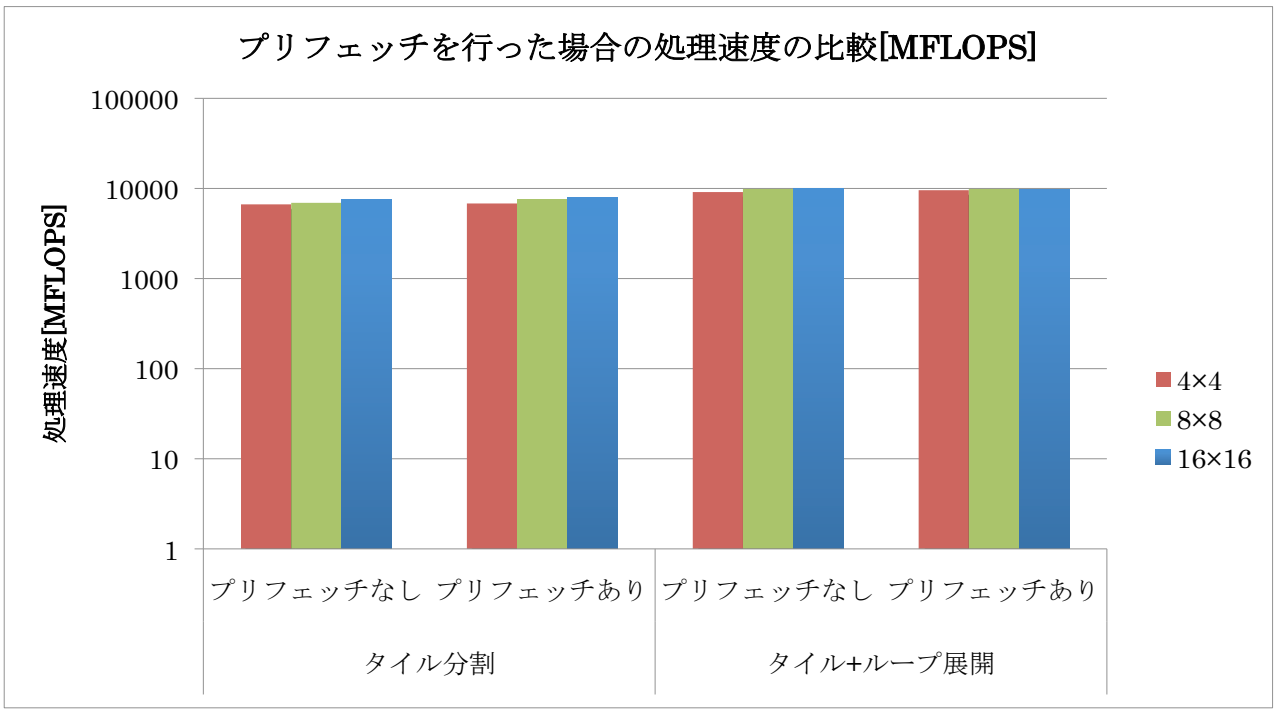


図 17 プリフェッチを行った場合の処理速度の比較

3.3.5 粒度の調整

前述したタイル分割の場合で、1 スレッドに複数の要素を求めさせるようにする。今回は、1 スレッドにつき 1 個の要素、2 個の要素、4 個の要素を求めさせる処理を、それぞれループ展開を行っている処理と行っていない処理で比較した。

表 7 と図 18 が示すように、粒度を調整して行列乗算を行った結果、1 スレッドあたりに計算させる要素の数を増やすに連れて、処理速度は全体的に上昇することが確認された。しかし、1 スレッドあたりに求めさせる要素を 1 から 2 に増やした時に比べ、2 から 4 に増やした時は速度の上昇が緩やかであり、場合によっては速度が低下したものもあった。タイルを読み込む量が増えた分メモリアクセスが増えたことと、1 スレッドあたりの処理の量が多くなりすぎたためと思われる。また、1 ブロックあたりのスレッド数を 16×16 で行った結果、要素数を 4 つ求める場合では、シェアードメモリ不足で処理を行うことができなかった。1 スレッドに対する演算量を増やすことで処理を高速化させることは可能だが、メモリ不足などが発生することから、この高速化の処理は、演算に使用する GPU のメモリ容量などのスペックを考慮することが必要である。

表 7 粒度の調整を行った際の処理速度の比較[MFLOPS]

	タイル			タイル+展開		
スレッド数	1×1	1×2	1×4	1×1	1×2	1×4
4×4	6,659.77	7,409.00	7,502.81	9,118.67	10,046.24	10,222.22
8×8	6,925.76	7,697.74	7,697.74	9,878.76	9,878.76	8,467.46
16×16	7,502.81	7,697.74	メモリ不足	10,021.05	9716.77	メモリ不足

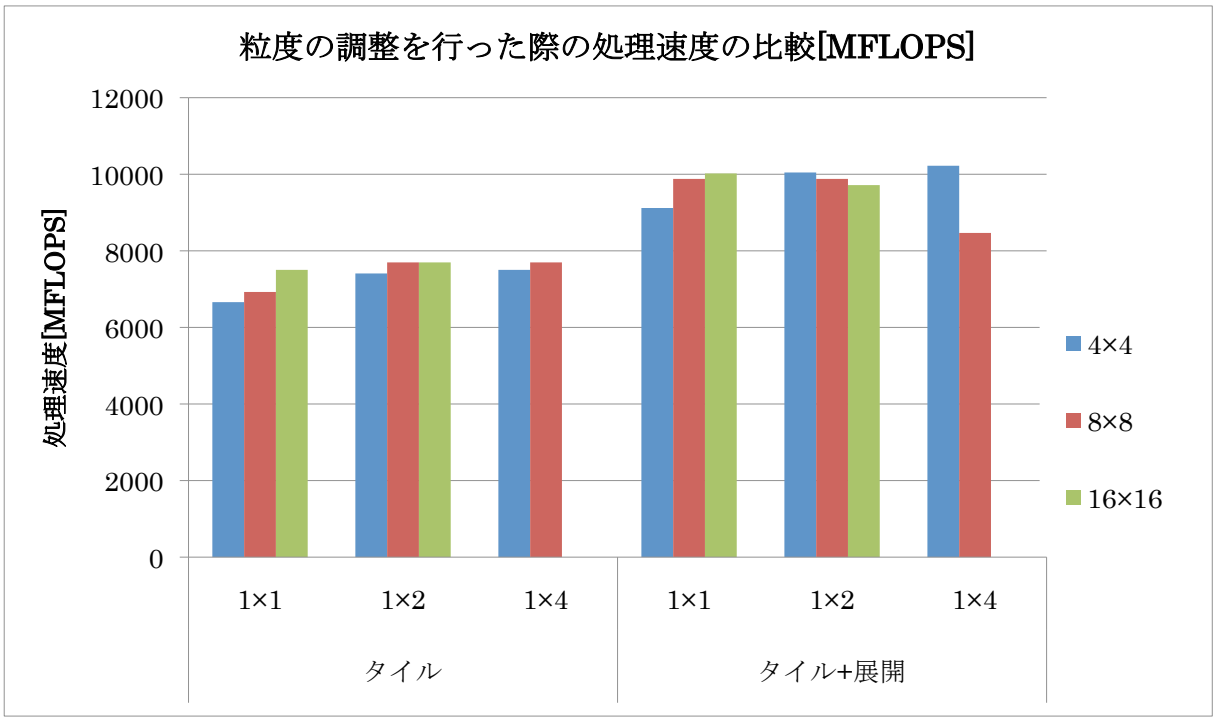


図 18 粒度の調整を行った際の処理速度の比較

3.3.6 全ての手法を混合させて処理を行った結果

ここれまでに検討したタイル分割、ループ展開、プリフェッチ、粒度の調整を混合させて総合的に処理速度を比較した。

今回の実験で判明したことは、

①表 8 および図 19 から、最も高速に処理できたのは 1 スレッドで 4 つの要素を計算し、プリフェッチなしで繰返し文を展開させた場合である。その速度を CPU による逐次処理と比較すると、約 200 倍に処理時間が短縮された。

②それぞれの貢献度は、コアレッシング、タイル分割、ループ展開、スレッドあたりの粒度の調整、プリフェッチの順で大きいという結果になった。

③プリフェッチ、およびスレッドあたりの粒度の調整は単体で行わせると通常よりも高速になる事が多かったが、ほかの手法と組み合せた場合、処理が遅くなる事もあった。

④グローバルメモリのみを使用した場合に比べてタイルを利用した場合はスレッドを増やしてもそこまで早くならないことが多かった。

コアレッシングを行わない場合、逆に CPU 処理よりも時間がかかってしまう場合があることから、コアレッシングには最大限の注意を払う必要がある。シェアードメモリを利用して処理を行わせると、グローバルメモリのみの場合よりも高速化に成功した。この事から、シェアードメモリやレジスタをうまく利用し、グローバルメモリへのアクセスを極力減らしていく事が重要である。なお、粒度を調整して処理を行なう場合はスレッド数を 16×16 で行うと、シェアードメモリの容量不足で正しい処理ができなかった。この事からメモリの容量やアクセス方法を理解し、どの処理を行わせればよいかを熟考していく事が必要である。

表 8 全処理を混合させて行った処理速度の比較[FLOPS]

プリフェッチなし						
スレッド数	タイル			ループ展開		
	1×1	1×2	1×4	1×1	1×2	1×4
4×4	6659.77	7409.00	7502.81	9118.67	10046.24	10222.22
8×8	6925.76	7697.74	7697.74	9878.76	9878.76	8467.46
16×16	7502.81	7697.74	メモリ不足	10021.05	9716.77	メモリ不足
プリフェッチあり						
スレッド数	1×1	1×2	1×4	1×1	1×2	1×4
	タイル			ループ展開		
4×4	6812.91	7599.02	7697.74	9560.01	9716.77	9716.77
8×8	7599.02	7799.05	7056.11	9878.76	9878.76	9261.19
16×16	7876.86	7056.11	メモリ不足	9716.77	8980.48	メモリ不足

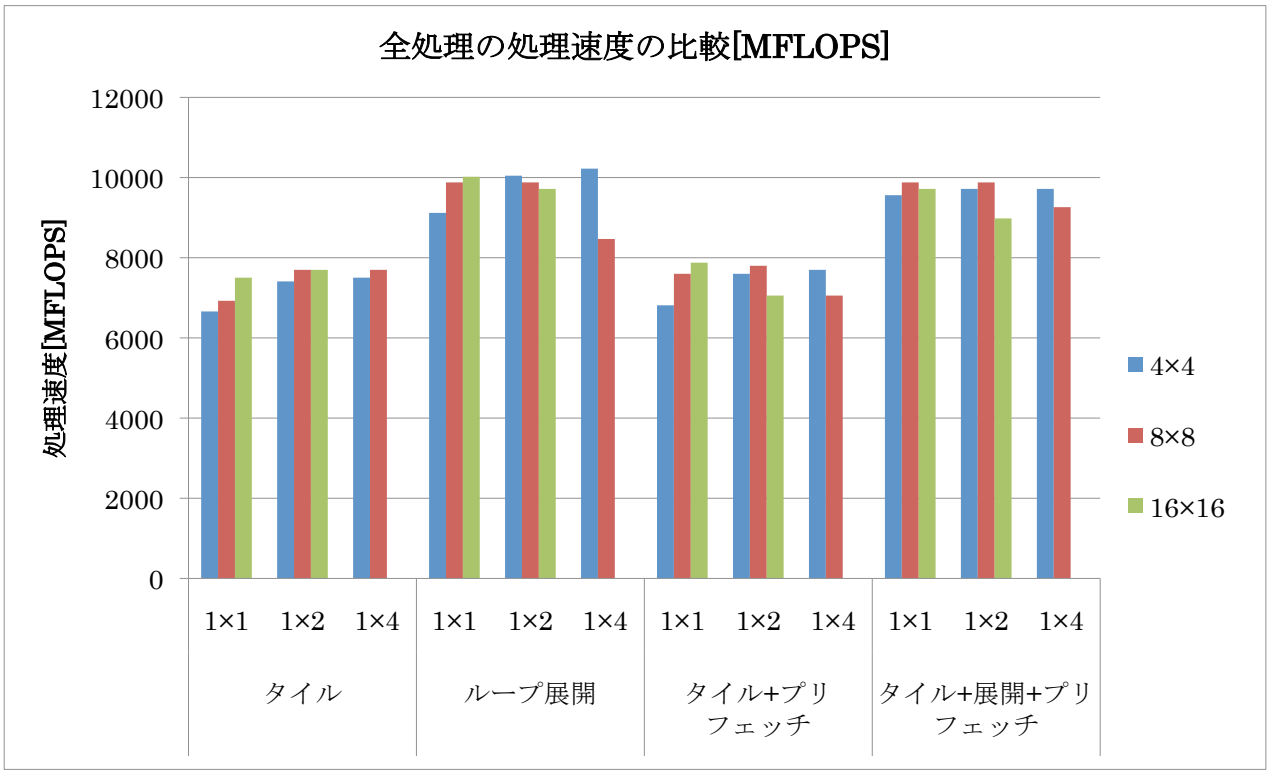


図 19 全処理の処理速度の比較

4.GPU の画像処理への応用

この章では GPU の並列処理能力をさまざまな画像処理に使用し、どのような手法を用いることで、どれほどの高速化が可能になるかを研究した。

実験ではグローバルメモリのみを使用した実験のほかに、コアレッシング、シェアードメモリによるタイル分割、ループ展開を使用した。なお、スレッドの粒度の調整の処理は今回の画像処理に向いていないと判断した。画像処理の際に RGB 画像を処理する際にシェアードメモリを画素の赤、緑、青の 3 原色分の領域分だけ確保する必要があるため、シェアードメモリ不足に陥りやすく、1 つのスレッドで複数の要素の出力を行う事は困難なためである。プリフェッチに関しては今回の実験では繰り返し処理が必要な動的計画法によるシームの検出の場合のみに行った。

4.1 空間フィルタ処理への利用

4.1.1 ラプラシアンフィルタの絶対値

ラプラシアンフィルタは、注目画素とそれに隣接する画素の輝度値の 2 次微分を計算することで、エッジの強調を行う線形空間フィルタである。このフィルタリングに GPU を利用する場合のプログラミング手法を研究した。

図 20 はラプラシアンフィルタによる空間フィルタ処理の概念を図示したものである。ラプラシアンフィルタは、注目画素とその 8 近傍にある画素の画素値の差の和を求める演算である。今回は、さらにその絶対値を求めることで、その結果を後述するシームカービングに利用する。ラプラシアンフィルタの絶対値を計算した画像は、図 20 の右の画像のようなエッジが強調された画像になる。

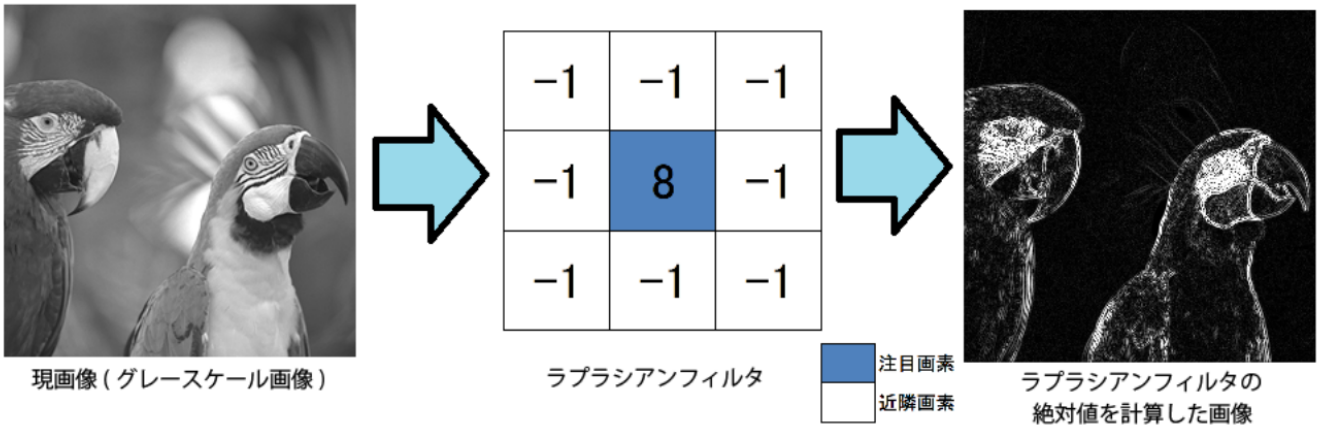


図 20 ラプラシアンフィルタの絶対値

4.1.2 ラプラシアンフィルタの絶対値の計算手法と実験結果

CPU による逐次処理では、1 画素ずつラプラシアンフィルタによる演算処理を行った。GPU による並列処理では、画素ごとの処理をスレッドに割り当てた。設定したブロック数に画像を分割し、それぞれの小領域に 1 つのブロックを割り当てた。そして、1 ブロック内のスレッド数だけ並列演算処理を行わせた。このとき、ブロックの総数は画像の画素数/割り当てられたスレッド数となる。

図 21 は X×Y 画素の画像の処理を、逐次処理で行う場合と並列処理で行う場合の違いを表している。逐次処理では(0,0),(1,0),...(X,0),(0,1),(1,1),...(X,Y)の画素をラスタ順に逐次的に処理する。並列処理では画像を N×M のブロックに分割し、それぞれの小領域をブロックに割り当て、小領域内のピクセルをスレッドに割り当てる。実験対象として、256×256 画素のグレースケール画像を使用し、次の場合について比較した。

① CUDA ファイルで CPU のみを利用した演算
② グローバルメモリのみを使用した GPU 演算
③ シェアードメモリを利用してタイル分割した GPU 演算

②～③については、スレッドブロックのスレッド数を 4×4、8×8、16×16 と 3 回ずつ行う。

これらの処理時間の比較を行った。

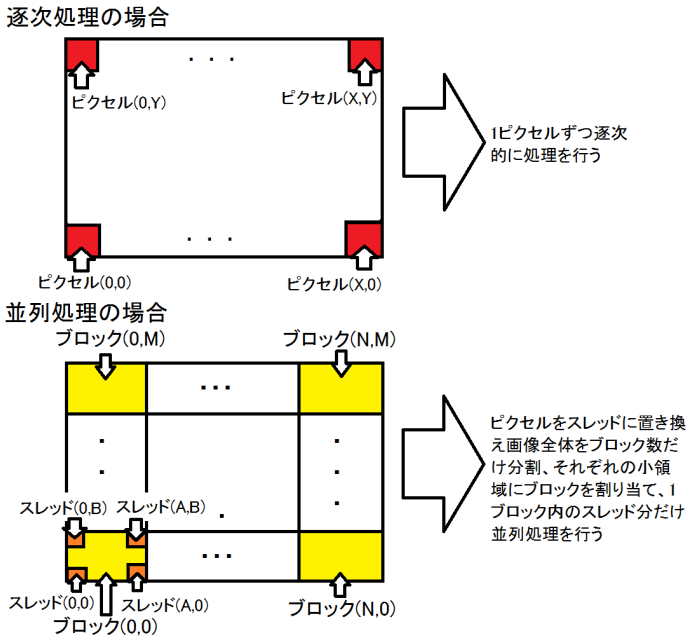


図 21 ラプラシアンフィルタにおける逐次処理と並列処理

表 9 および図 22 にそれぞれの場合の処理時間を示す。並列処理と逐次処理の処理時間を比較した結果、逐次処理に比べグローバルメモリのみを使用した場合は約 5 倍、シェアードメモリを使用してタイル分割を行った場合は約 30 倍に高速化された。グローバルメモリのみ使用時にさほど高速化できなかった理由は、演算に必要な画像値を取得する時、毎回 9 画素分をグローバルメモリにアクセスしなければならないのに対し、グローバルメモリから得たデータを使用した演算回数が 9 回であることから、CGMA 率が 1.0 と効率が悪いためである。シェアードメモリを利用した場合、各スレッドが画素の値をシェアードメモリに書き込み、先述のアクセスの部分でシェアードメモリにアクセスすれば、各スレッドがグローバルメモリへのアクセス回数が 1 回、浮動小数演算の回数が 9 回となり、CGMA 率が 9.0 と演算の割合が大きくなるため、処理速度が改善された。

この事から、1 つの配列の要素を複数回必要とする場合に、シェアードメモリが活用することが有効である。

表 9 ラプラシアンフィルタの処理時間[ms]

スレッド数 ④	並列		逐次①
	通常②	シェアードメモリ使用 ③	
4×4	0.227	0.030	0.991
8×8	0.194	0.034	0.983
16×16	0.187	0.032	0.990

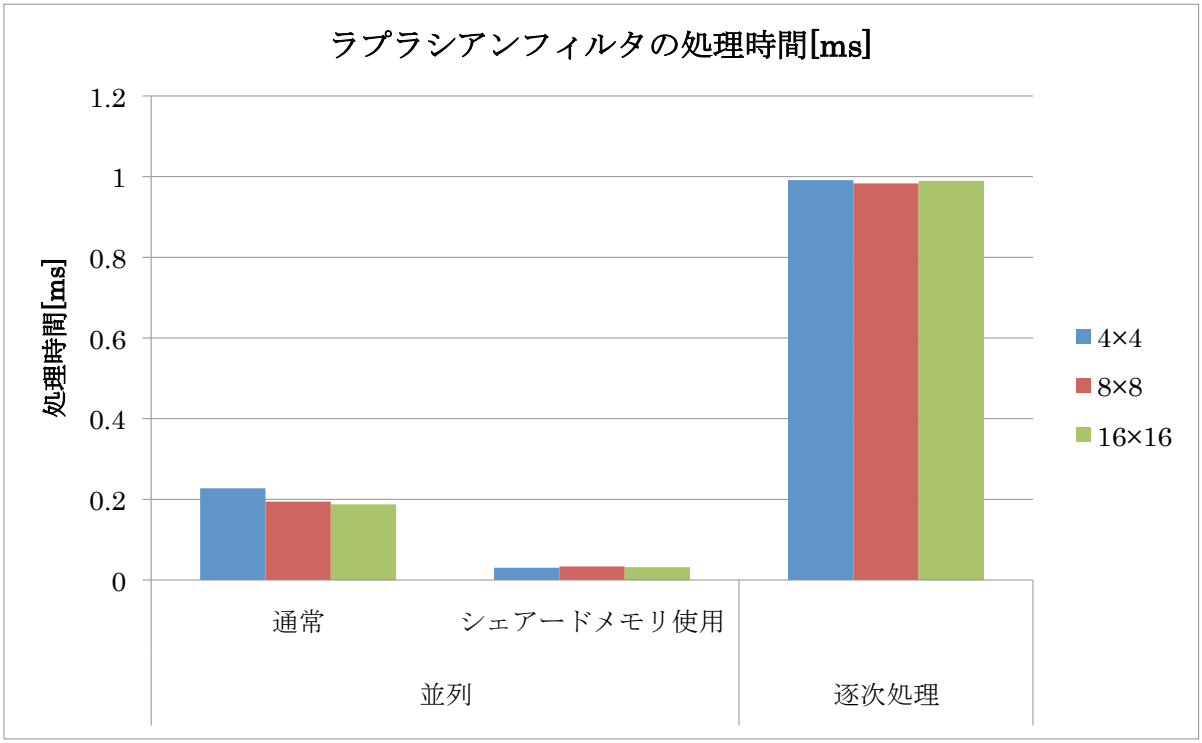


図 22 ラプラシアンフィルタの処理速度の比較

4.2 動的計画法によるシームの検出への利用

4.1 で生成したラプラシアン画像を使用した画像のリサイズ技術の一つにシームカービング[4]がある。シームカービングで行われる動的計画法による最適シームの計算処理を並列化させ、処理時間の高速化を実現した。

4.2.1 シームカービング

シームカービングは画像を拡大・縮小するリサイズ技術のひとつで、画像内の人間の視覚に重要な部分を判断し、その情報を用いて、画像に不自然なゆがみが生じないように変形することができる。

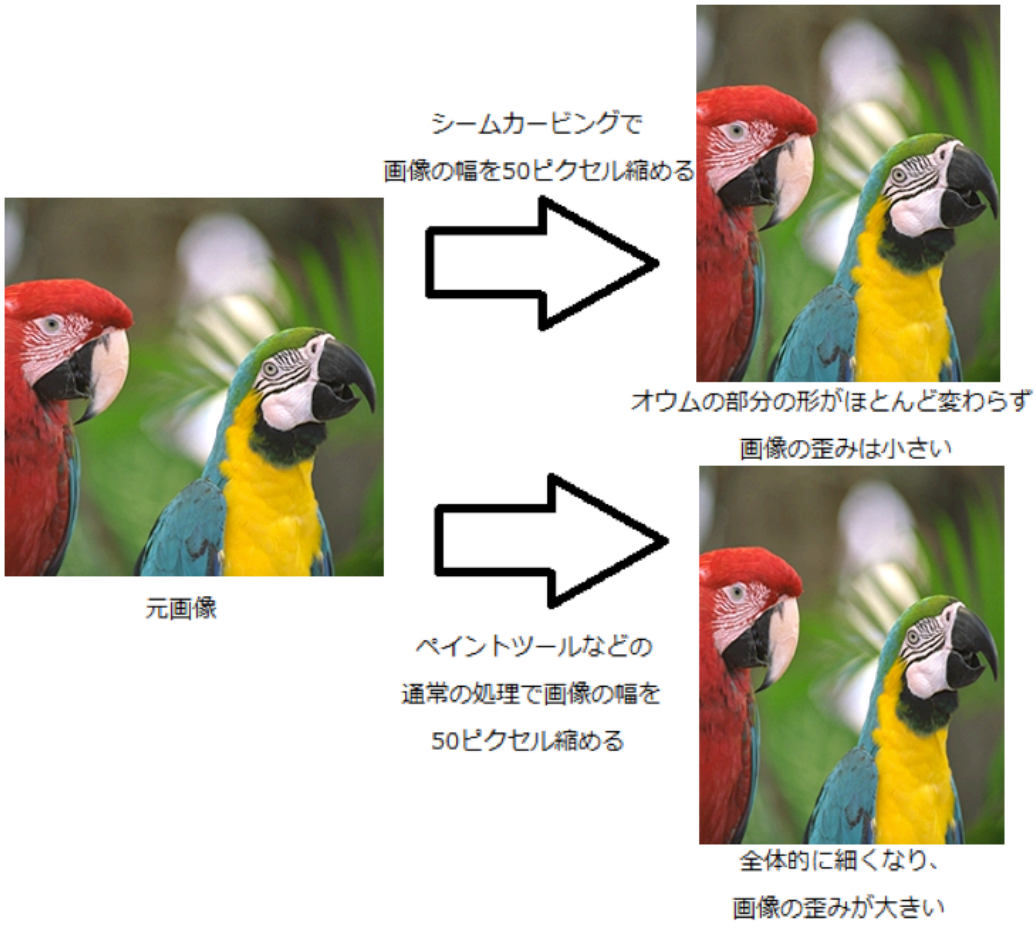


図 23 シームカービング

図 23 はシームカービングによって画像の幅を縮めた場合とペイントツールなどの通常の処理で画像の幅を 50 ピクセル縮めた場合の比較を図示したものである。通常の処理で縮めた場合に対し、シームカービングで処理した場合はオウムの絵に歪みが少ない状態で画像が縮められている。

この実験では図 24 に表すように削除すべき画素のつなぎ目であるシームを求める処理を並列化した。

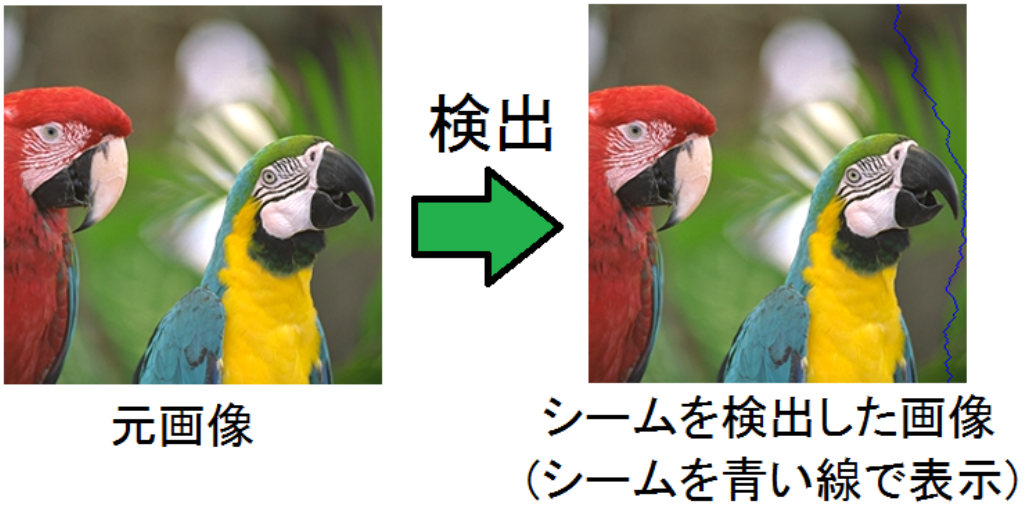


図 24 シームの検出

削除に最適なシームを検出するために動的計画法が利用される。動的計画法は「制約式が逐次的であるとき、原問題を部分問題群に埋め込み、各部分で最適な答えを見つけ出す。これを逐次的に行い、最終的に全ての答えのうち最適な答えを導き出す」手法である。

この実験では、前述のラプラシアンフィルタで求めた画像の画素値の大きさを、視認性に関係するエネルギと解釈して利用する。逐次的なアルゴリズムは次のようになる。

1：各画素にエネルギ(E(i,j))を定義する。エネルギ値として、ラプラシアンの絶対値を用いる。

2：画像サイズと同じ大きさの表（DP(i,j)）を定義し、その初期値を 0 とする。

3:DP の第一行に E の第一行をコピーする

4:DP の第 2 行から最終行まで、5 を繰り返す

5:DP の注目行(i)について、第一列から最終列まで 6 を繰り返す

6:DP の注目画素(i,j)のエネルギ E(i,j)と DP(i-1,j-1)(注目画素の左上の画素),DP(i-1,j)（注目画素の上の画素),DP(i-1,j+1)（注目画素の右上の画素)の和の中で最小になるものを DP(i,j)とする。

7：（ここに達した時、DP の全ての値が計算されている）DP の最終行の中で、最も値が小さいものを選択し、そこから DP を上にたどっていくことで、エネルギが最小になるシームを検出する。

この動的計画法を利用してシームを検出する方法の概要を図 25 に図示した。

本実験では、この中の 2～6 までの計算手順について並列化を検討した。

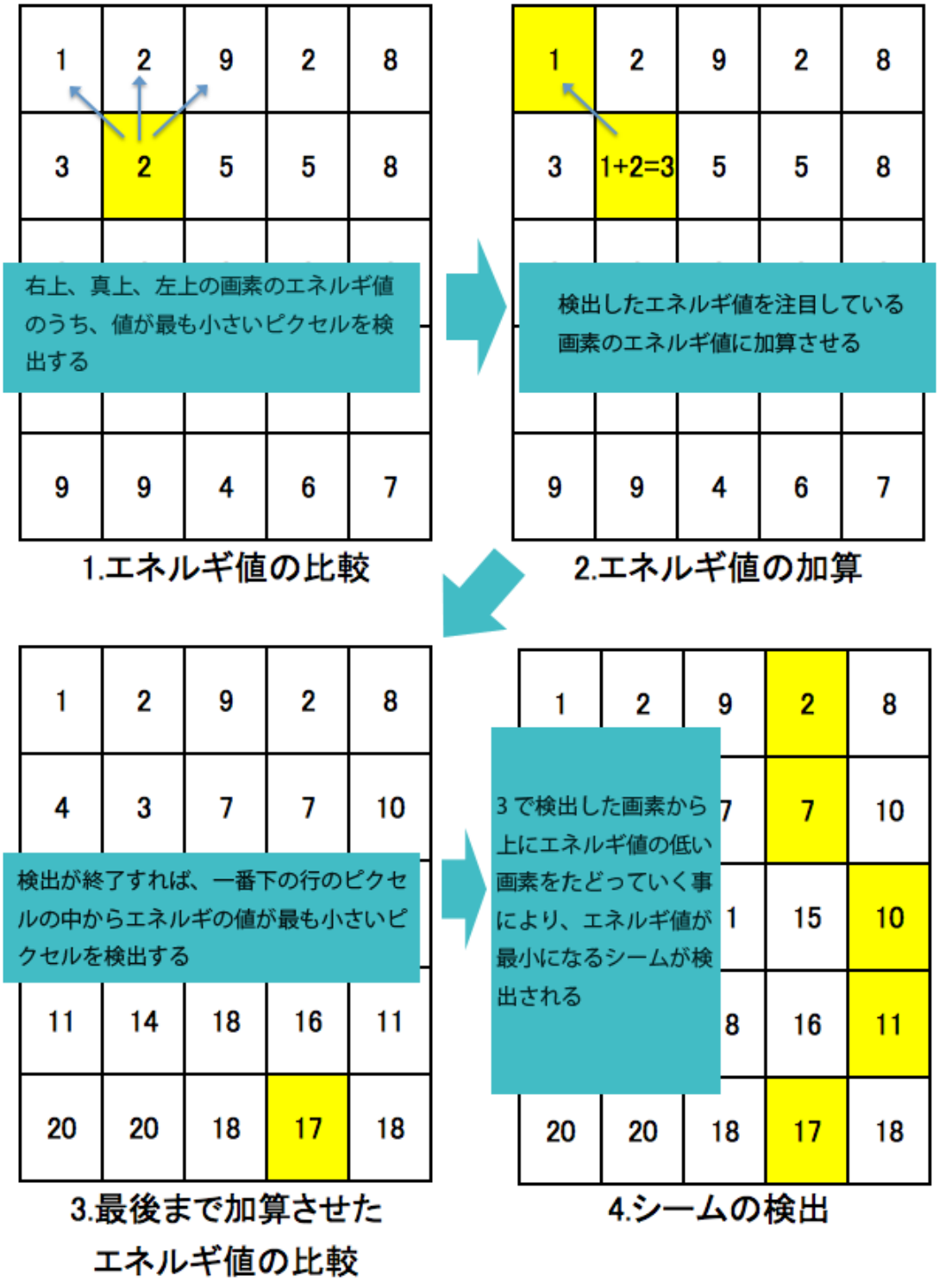


図 25 動的計画法によるシームの検出

4.2.2 シームカービングの計算手法と実験結果

CPU で逐次処理行う場合はラプラシアンフィルタ処理同様、1 画素ずつ逐次的に求めていく。

GPU で並列処理を行う場合、スレッド間で連動しながら処理を行う必要がある。ラプラシアンフィルタのように、全スレッドが独立して計算処理を行うのではなく、縦 1 列分の画素列を 1 スレッドに処理させ、繰り返し処理の 1 回あたりにつき画像の横 1 行分のエネルギを求めさせる。図 26 は動的計画法を並列処理で行わせるときの概要である。図 25 で行う処理を各スレッドに画像の 1 行分行わせた。

比較対象として、

① CUDA ファイルで CPU のみを使用した処理

② グローバルメモリのみ使用した処理

③ 1 ブロック分をシェアードメモリに書き込み、その値を使用する処理

④ ③の処理に加え次にエネルギを求める行をプリフェッチさせる処理

②～④については、スレッドブロックのスレッド数を 4×4、8×8、16×16 と 3 回ずつ行う。

これらの処理時間の比較を行った。

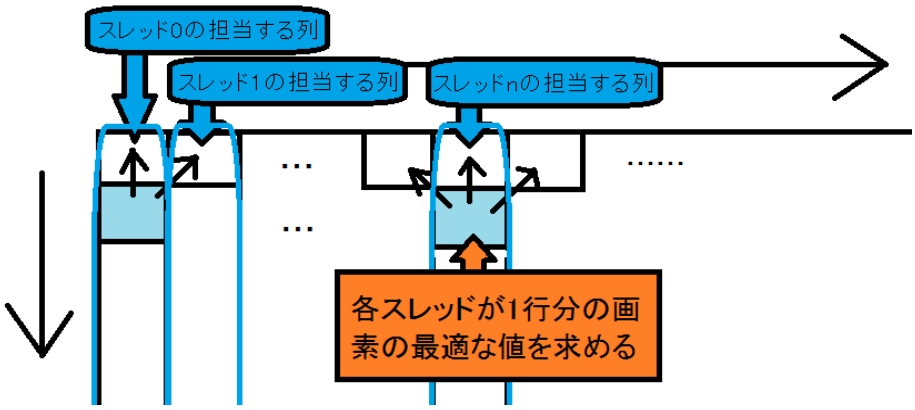


図 26 並列処理で実装した動的計画法

表 10 および図 27 にそれぞれの場合の処理時間を示す。逐次処理に比べ、並列処理では約 90 倍の高速化に成功した。ラプラシアンフィルタはグローバルメモリへのアクセス数が画素あたり 9 回と多い上に CGMA 率が 1:1 と効率が悪く、しかも、それらは 3 行に分かれているので、完全にコアレッシングすることができない。それに対し、動的計画法では表の一つの項目を計算するためのメモリアクセスが 3 回であり、浮動小数演算回数は 4 回となっている。このため、CGMA 率は 1.3 とラプラシアンフィルタに比べ高い。また、グローバルメモリへのアクセス回数そのものも少ないため、メモリアクセスにかかる時間がラプラシアンフィルタに比べ少ない。また、プリフェッチによる処理を行ったところ約 0.001ms ほどの処理時間の短縮が行えた。グローバルメモリからシェアードメモリへデータを移す際に、独立した処理を加える事によって、わずかながらにレイテンシを隠蔽する

事が出来た。

この実験ではグローバルメモリのみを使用したときと、シェアードメモリを使用したときではほとんど差がなかった。このときグローバルメモリへのアクセス回数が 2 回、浮動小数演算回数が 4 回となっているため CGMA 率は 2.0 と増えているものの 1 つの要素が使われる回数がもともと少なく、シェアードメモリによる高速アクセスの恩恵を得られなかったためだと思われる。

表 10 動的計画法によるシームカービングの処理時間[ms]

スレッド数	並列処理			逐次処理①
	グローバルメモリのみ②	シェアードメモリ使用③	タイル+プリフェッチ④	
4×4	0.0163	0.0127	0.0126	1.1412
8×8	0.0159	0.0126	0.0126	1.3307
16×16	0.0154	0.0126	0.0125	1.1633

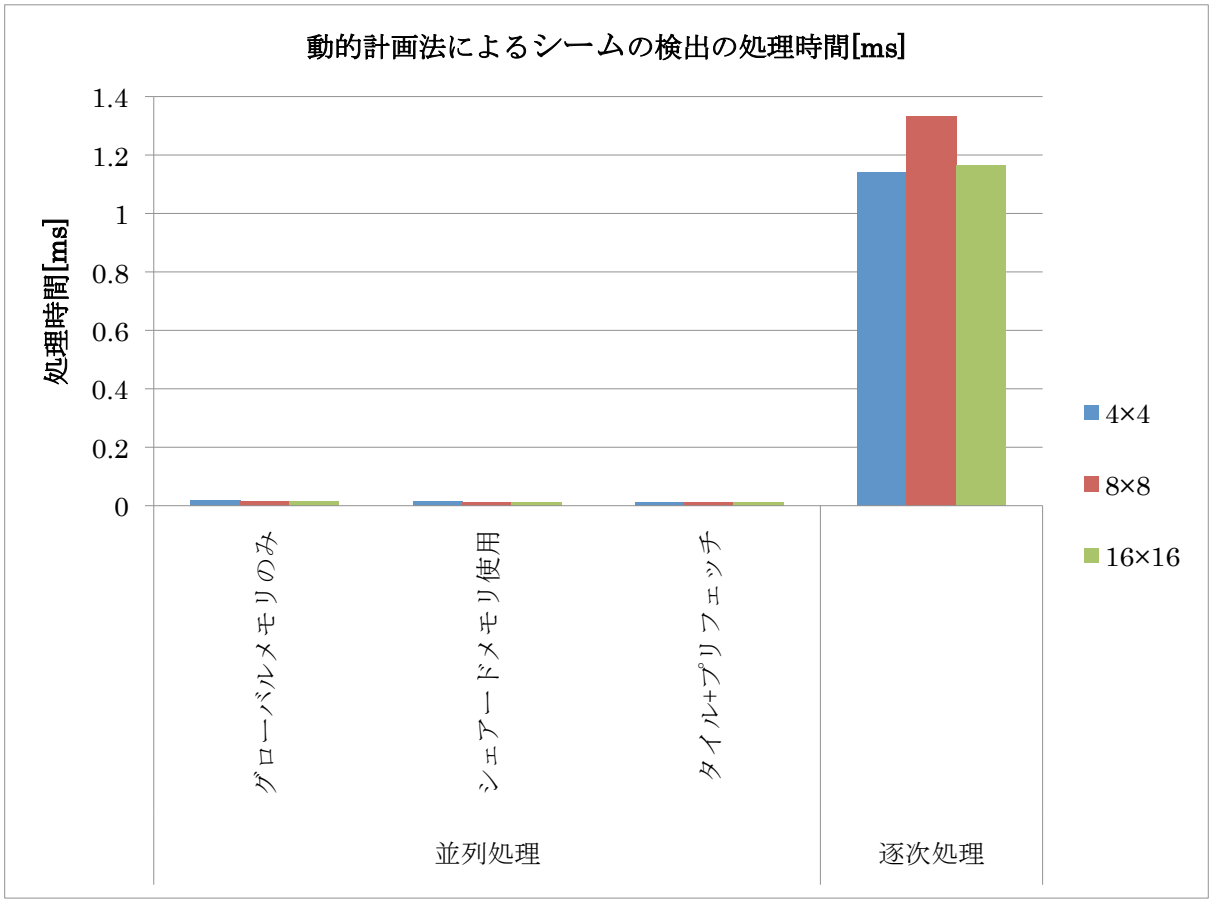


図 27 動的計画法の処理時間の比較

4.3 ライトフィールドレンダリングにおけるリフォーカス画像の生成への利用

4.3.1 リフォーカス画像の生成

ライトフィールドレンダリングは、三次元空間の光線情報を記録したライトフィールドを用いて任意視点映像を生成することや、ピントが異なった写真画像を生成する技術である。本論文では、代表的なライトフィールドレンダリングである、リフォーカス画像の生成に GPU を利用することを検討した。

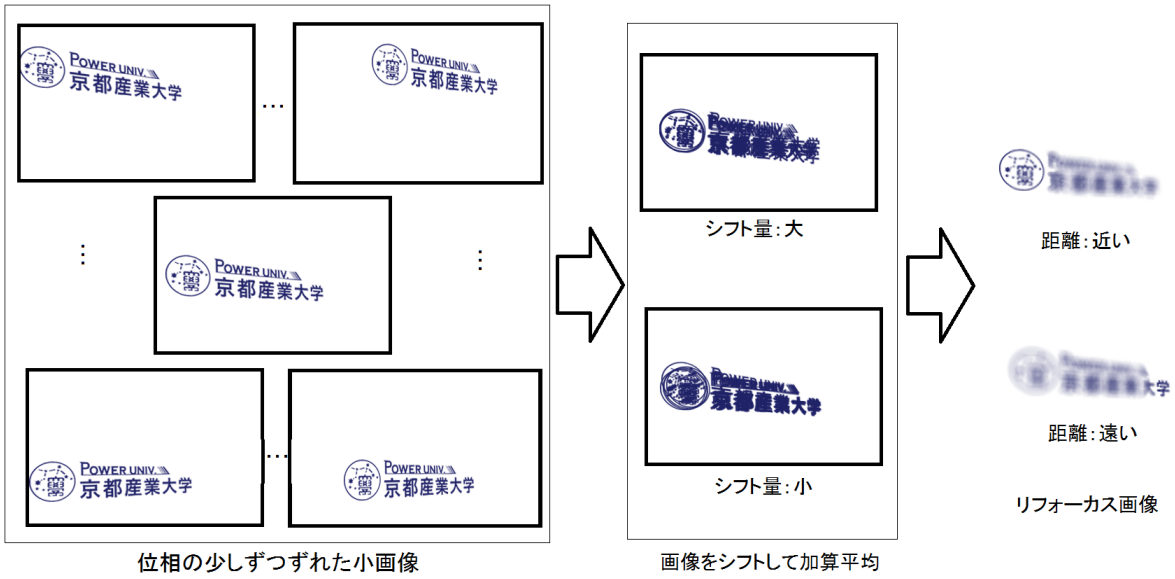


図 28 リフォーカス画像の生成の手法

図 28 にリフォーカス画像を生成するアルゴリズムの概要を示す。レンダリングに使われる元画像は、多数のステレオ画像群である。これらは視差によって少しずつ被写体の位置がずれており、そのずれの大きさは被写体とカメラとの距離によって違う。これを重ね合わせるために小画像を適量平行移動して加算平均させる。これにより、移動量に応じた奥行きに存在する被写体の位置が合うので、シャープな画像を生成できる。逆にそうでない被写体の像はぼやける。

4.3.2 リフォーカス画像の計算手法と実験結果

今回の実験では、図 29 に示した視点が少しずつ異なる 720×480 画素の 7×7 枚の画像を用いて、カメラからの距離別にピントが異なる 30 枚の画像を形成するまでの処理時間を比較する。レンダリングしたリフォーカス画像の例を図 30 に示す。

計算手法としては、まず、すべての小画像の画素の位置を視差に応じて平行移動させる。次に平行移動させた全ての小画像の画素ごとに加算平均を行い、その平均値を出力画像の画素値として出力させる。この処理を出力画像の枚数分だけ繰り返し処理を行う。逐次処理と並列処理で、この出力画像の全ての画素値を導きだすまでの処理時間の比較を行う。

CPU による逐次処理では、画像を平行移動させる処理と、それらを加算平均することで出力画像を求める処理を、1 画素ずつ逐次的に行う。この処理を出力画像の枚数分だけ行う。

GPU による並列処理では、平行移動させた画像の画素値を配列に代入し、ブロック分だけ小領域にわけ、1 スレッドあたり 1 画素ずつ並列処理で求めさせる。続いて加算平均による出力画像の画素値も別の並列処理で行わせる。こちらも、出力画像を配列に置き換え、ブロック分だけ小領域に分け、 1 スレッドあたり 1 画素分加算平均の処理を行わせる。比較対象として、

① CUDA ファイルで CPU のみを使用した処理
② グローバルメモリのみ使用した処理
③ 1 ブロック分をシェアードメモリに書き込み、その値を使用する処理
②～③については、スレッドブロックのスレッド数を 4×4、8×8、16×16 と 3 回ずつ行う。これらの処理時間の比較を行った。

図 29 実験に用いた画像

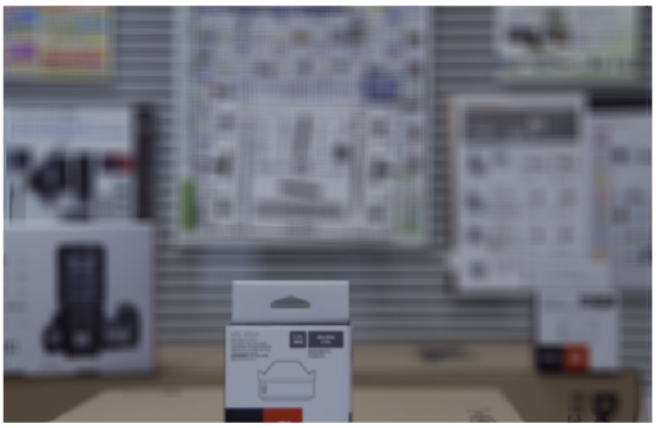


図 30 リフォーカス画像の例

表 11 および図 31 にそれぞれの場合の処理時間を示す。逐次処理に比べて GPU を用いた並列処理を行った場合、約 80 倍の処理速度になった。今回の実験で逐次処理を行った場合、720×480 画素の 49 枚のカラー画像から 30 枚の画像を生成するため、約 40 秒という長い処理時間になった。これを並列処理で行った場合、シフトさせた画像の配列の生成と出力画像の生成で並列処理を行う。この時、グローバルメモリへのアクセス回数は 4 回、浮動小数演算回数は 8 回となり、CGMA 率が 2.0 となるため、ラプラシアンフィルタや動的計画法によるシームの計算と比較して、並列処理をより効率化できた。一方、シェアードメモリによるタイル内での処理を行いグローバルメモリへのアクセスを極力減らした場合、CGMA 率は 4 倍増加したものの、グローバルメモリのみ使用した場合で既に高い高速化が行えたためか期待したほどの高速化は達成出来なかった。

表 11 ライトフィールドレンダリングの処理時間[sec]

スレッド数	並列処理		逐次処理①
	グローバルメモリのみ②	シェアードメモリ使用③	
4×4	1.057	0.976	40.366
8×8	0.712	0.653	40.266
16×16	0.542	0.505	40.290

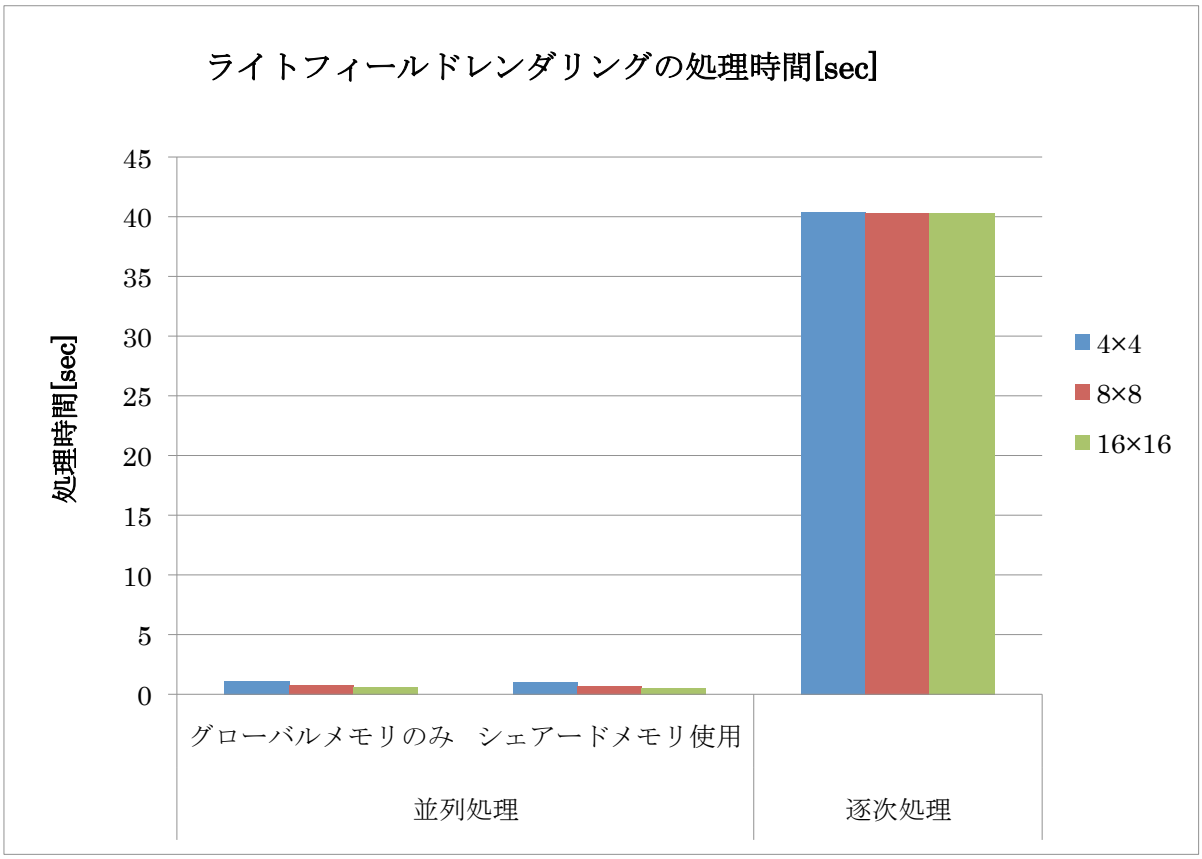


図 31 ライトフィールドレンダリングの処理時間の比較

5.結論

GPU を有効に使うためのプログラミング手法として、コアレッシング、タイル分割、ループ展開、プリフェッチ、スレッドの粒度の調整を実装して、その効果を行列演算で確認した。ついで、それらの手法を 3 種類の画像処理に応用した。その結果、行列演算では最大約 200 倍の高速化、ラプラシアンフィルタでは最大約 30 倍の高速化、動的計画法によるシームの検出では約 90 倍の高速化、リフォーカス画像の生成では最大約 80 倍の高速化を達成した。

この研究から、画像処理に GPU を利用する事は、非常に有効であることを実証した。今回の研究で、GPU による並列処理能力の高さを知る事ができた。画像処理の場合は最大約 80 倍の高速化に成功したが、行列演算が 200 倍の高速化を達成していることから、メモリのアクセス方法やブロックあたりのスレッド数などの調整を考えたりし、工夫を加える事によってもっと速くできると感じた。

まだまだシェアードメモリなどの使用法に改善の余地は見られるものの、GPU を使用して画像処理を高速化させるという目標を達成し、どういった処理にどういった手法を使用するかを導くという事も発見出来た。

現在でも GPU や GPGPU の研究は進んでおり、CUDA だけでなく OpenACC といった開発環境も開発されており、どんどん GPU の性能を最大限に、かつ簡単に活かせる研究が行われるものと思われる。

謝辞

本研究の遂行にあたり、終始適切な助言を賜り、また丁寧に指導してくださった蚊野浩教授に深謝いたします。また、多くの情報やアドバイスをいただき、時に励まし合い、互いに支え合った蚊野研究室の同期･後輩の皆様に感謝いたします。最後に私の学生生活を支えてくださり，修士研究の機会を与えてくださった家族の皆様に感謝いたします。

参考文献

[1] 青木尊之, 額田彰, 「はじめての CUDA プログラミング」,工学社(2009 年)

[2] 高野芙美代,森吉達治, 「GPU を用いた動画像符号化の高速化」, 映像情報メディア学会誌 Vol.66, No.10, p 823~826(2012 年)

[3]市村直幸,「方向マップを用いた局所不変特徴量の抽出」, 映像情報メディア学会誌 Vol.66, No.10, p 831~834(2012 年)

[4] 山崎俊彦, 「100 行で書く画像処理最先端 画像の編集 : シームカービング(Seam Carving)」, 映像情報メディア学会誌 Vol. 64, No. 2, p 208～215(2010 年)

[5] David B. Kirk, Wen-mei W.Hwu, 「CUDA プログラミング実践講座」, ボーンデジタル(2010 年)

[6] 蚊野浩, 「マイクロレンズで 50 枚の画像画素の演算でピントを調整」, 日経エレクトロニクス 2012.8.20 no.1089 p44～47,日経 BP 社(2012 年)

付録

今回の実験で制作したプロジェクトの一覧

プロジェクト名	処理内容
Matrix	行列乗算を行うプログラムで、CPU による逐次処理、コアレッシングとタイル分割、およびループ展開のそれぞれの有無による処理速度の計測を行う
Matrix_2	上記の Matrix の処理において、タイル分割による処理を行うときに 1 スレッドあたりに行列の要素を 2 つ求めさせるようにしたプログラム
Matrix_4	上記の Matrix の処理において、タイル分割による処理を行うときに 1 スレッドあたりに行列の要素を 4 つ求めさせるようにしたプログラム
Matrix_Pre	上記の Matrix の処理において、タイル分割による処理を行うときにプリフェッチを行うようにしたプログラム
Mairix_Pre_2	上記の Matrix の処理において、タイル分割による処理を行うときにプリフェッチを行いながら 1 スレッドあたりに行列の要素を 2 つ求めさせるようにしたプログラム
Matrix_Pre_4	上記の Matrix の処理において、タイル分割による処理を行うときにプリフェッチを行いながら 1 スレッドあたりに行列の要素を 4 つ求めさせるようにしたプログラム
SeamDynamic	ラプラシアンフィルタの絶対値の計算と動的計画法によるシームの検出を行うプログラム
SeamDynamicG	上記の SeamDynamic の処理を GPU による並列処理で行うプログラム
Refocus	リフォーカス画像を生成するプログラム
RefocusG	上記の Refocus の処理を GPU による並列処理で行うプログラム