GPUを用いた画像処理の高速化

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回生
蚊野研究室　鳴海　弘太郎

研究の目的

- 画像処理は膨大なデータを規則的に処理する必要があり、計算コストが高い。
- GPUを用いたハイパーフォーマンスコンピューテングが成功を収めている。
- 本研究では、GPUを画像処理に利用する手法について検討した。

発表の内容

- GPU, CUDAの概要
- GPUの性能を最大化するプログムラム手法の提案
- 提案手法の画像処理への応用

発表の内容

- GPU, CUDAの概要
- GPUの性能を最大化するプログムラム手法の提案
- 提案手法の画像処理への応用

GPU(Graphics Processing Unit)

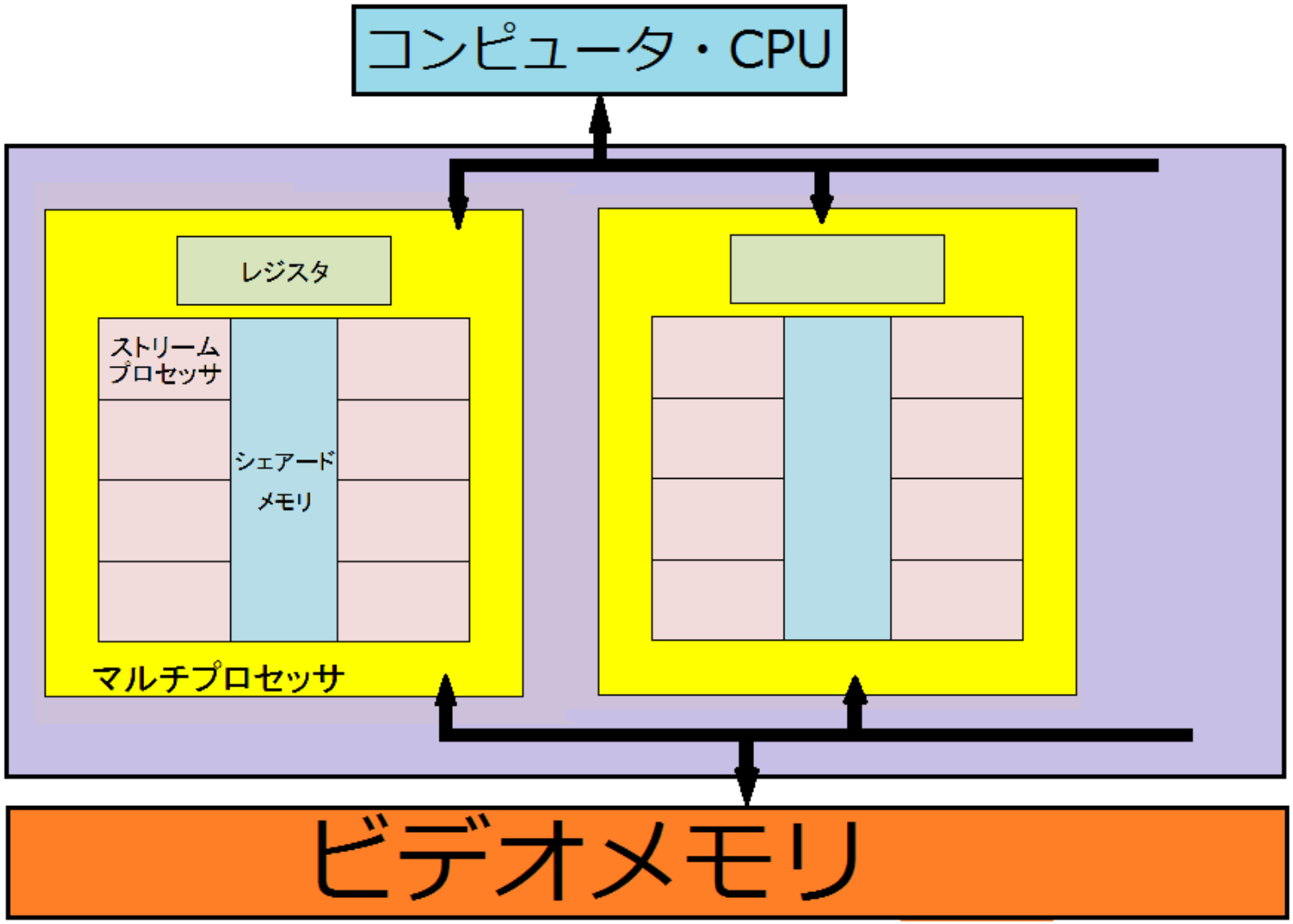
- 並列処理能力を備えた演算装置
- 多数のプロセッサを内蔵し、大量のデータを同時に処理する能力を備える。
- CG処理以外の用途に利用する技術をGPGPU(GPUによる汎用計算)と呼ぶ。

GPUの画像処理での実用例

- 動画像符号化の高速化
- 医療：MRIの画像解析
- 電子顕微鏡：分子の可視化

など

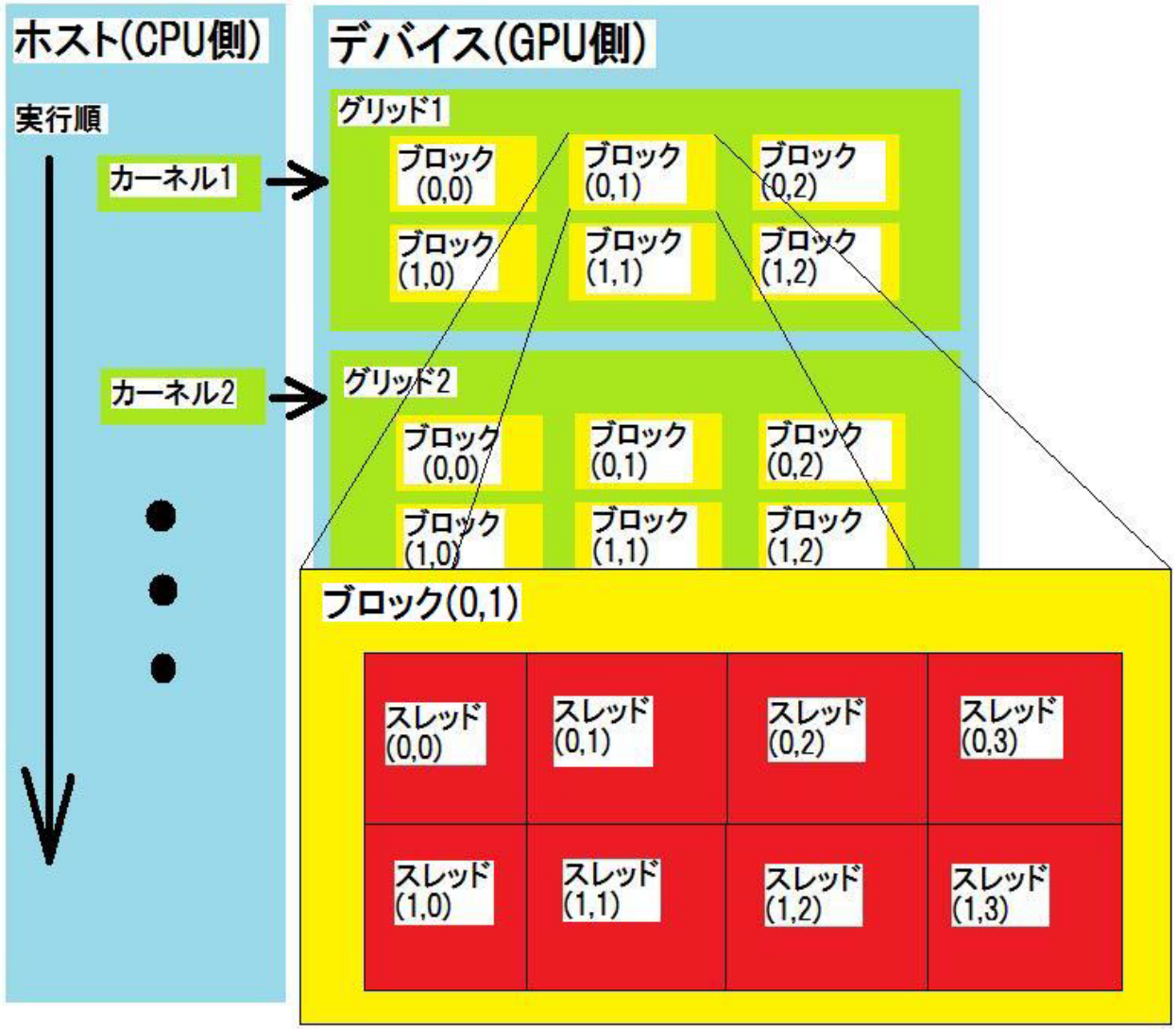
NVIDIA社製のGPUのアーキテクチャ



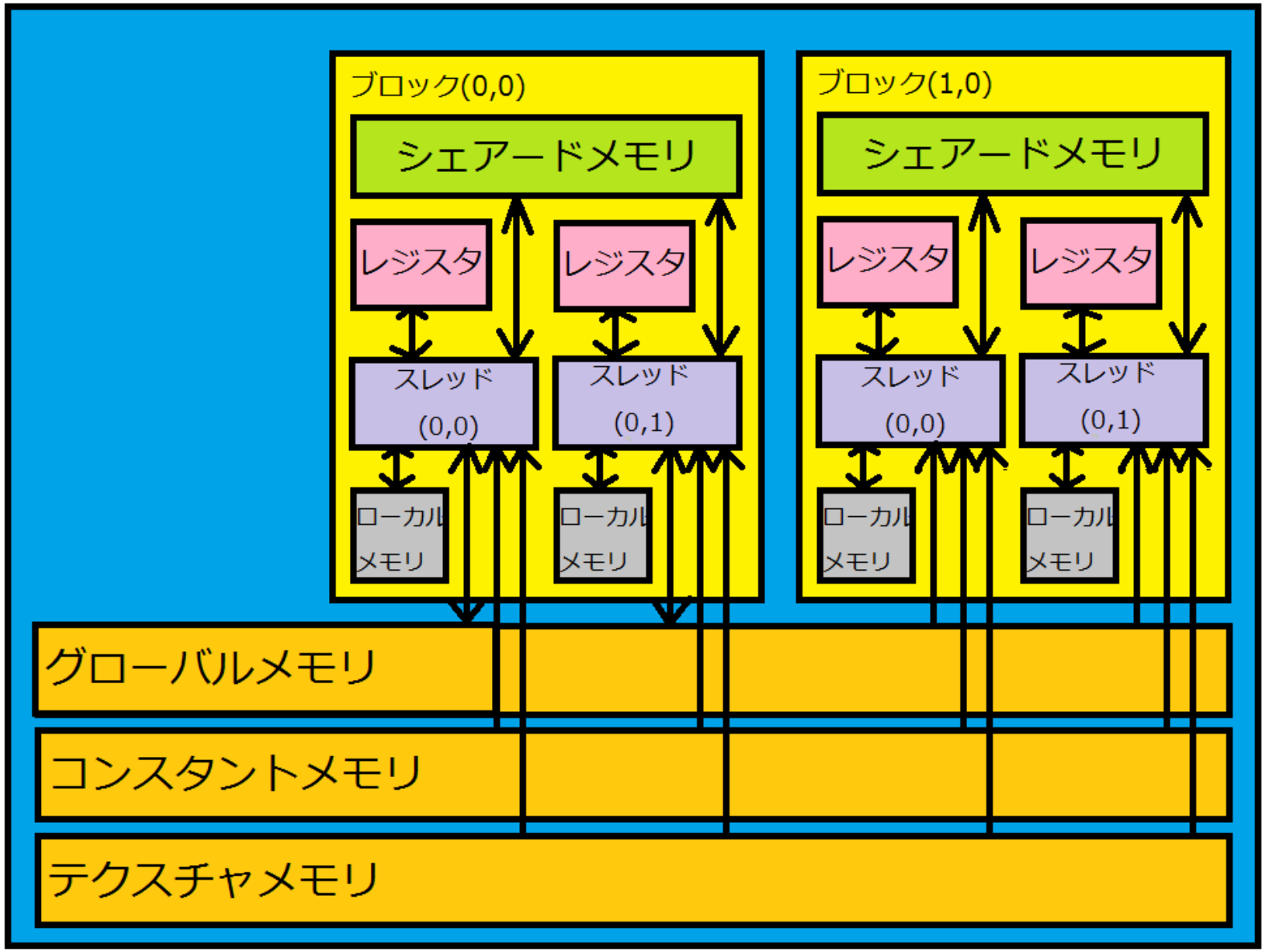
CUDA
（Compute Unified Device Architecture）

- NVIDIA社が提供するGPU向けの、C言語ベースの統合開発環境。
- 処理の最小単位をスレッドと呼び、スレッド数によって並列度が決まる。
- GPUのストリームプロセッサがスレッドに割り当てられる。

スレッド, ブロック, グリッド



CUDAのメモリモデル



発表の内容

- GPU, CUDAの概要
- GPUの性能を最大化するプログムラム手法の提案
- 提案手法の画像処理への応用

GPUの性能を最大化するプログラム手法

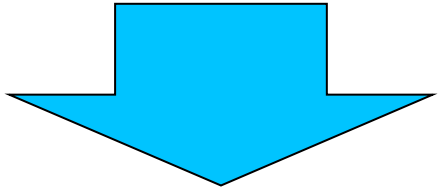
- コアレッシング
- メモリ階層の有効利用

GPUの性能を最大化するプログラム手法

- コアレッシング
- メモリ階層の有効利用

コアレッシング

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、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、高速にアクセスできる。



コアレスアクセス

コアレスアクセスを行わせることを
コアレッシングという

メモリ上の配列要素の並び

配列

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

グローバルメモリ上の並び方は…

(0,0)	(0,1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	…

配列の要素は行優先に並んでいる

コアレッシングされていないメモリアクセス

	各スレッドが1番目に読み込む要素	各スレッドが2番目に読み込む要素	各スレッドが3番目に読み込む要素	各スレッドが4番目に読み込む要素
スレッド0が読み込む要素	(0,0)	(0,1)	(0,2)	(0,3)
スレッド1が読み込む要素	(1,0)	(1,1)	(1,2)	(1,3)
スレッド2が読み込む要素	(2,0)	(2,1)	(2,2)	(2,3)
スレッド3が読み込む要素	(3,0)	(3,1)	(3,2)	(3,3)

各スレッドが同時に読み込む要素が隣り合っていない

コアレスアクセスがされておらず、メモリアクセスに遅延が起こる

コアレッシングされたメモリアクセス

スレッド0が読み込む要素	スレッド1が読み込む要素	スレッド2が読み込む要素	スレッド3が読み込む要素	
(0,0)	(0,1)	(0,2)	(0,3)	各スレッドが1番目に読み込む要素
(1,0)	(1,1)	(1,2)	(1,3)	各スレッドが2番目に読み込む要素
(2,0)	(2,1)	(2,2)	(2,3)	各スレッドが3番目に読み込む要素
(3,0)	(3,1)	(3,2)	(3,3)	各スレッドが4番目に読み込む要素

各スレッドが同時に読み込む要素が隣り合っている

コアレスアクセスがされている

GPUの性能を最大化するプログラム手法

- コアレッシング
- メモリ階層の有効利用

メモリ階層の有効利用

- 演算のたびに毎回グローバルメモリにアクセスするとレイテンシが発生。
- 要素をシェアードメモリなどの高速メモリにコピーしてから演算処理に使用する。
- 特定データを複数回利用する場合、各スレッドがグローバルメモリにアクセスする回数が減少する。

コアレッシングの有無による処理速度の違い

512×512の行列乗算を行った場合の処理速度

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	2,339.48	51.93
16×16	3,187.93	41.11

単位:MFLOPS(FLOPS:1秒当たりの浮動小数演算回数)

シェアードメモリを活用したときの処理速度の違い

512×512の行列乗算を行った場合の処理速度

スレッド数	シェアードメモリ使用	グローバルメモリのみ
4×4	6,659.77	872.28
8×8	6,925.76	2,339.48
16×16	7,502.81	3,187.93

単位:MFLOPS(FLOPS:1秒当たりの浮動小数演算回数)

発表の内容

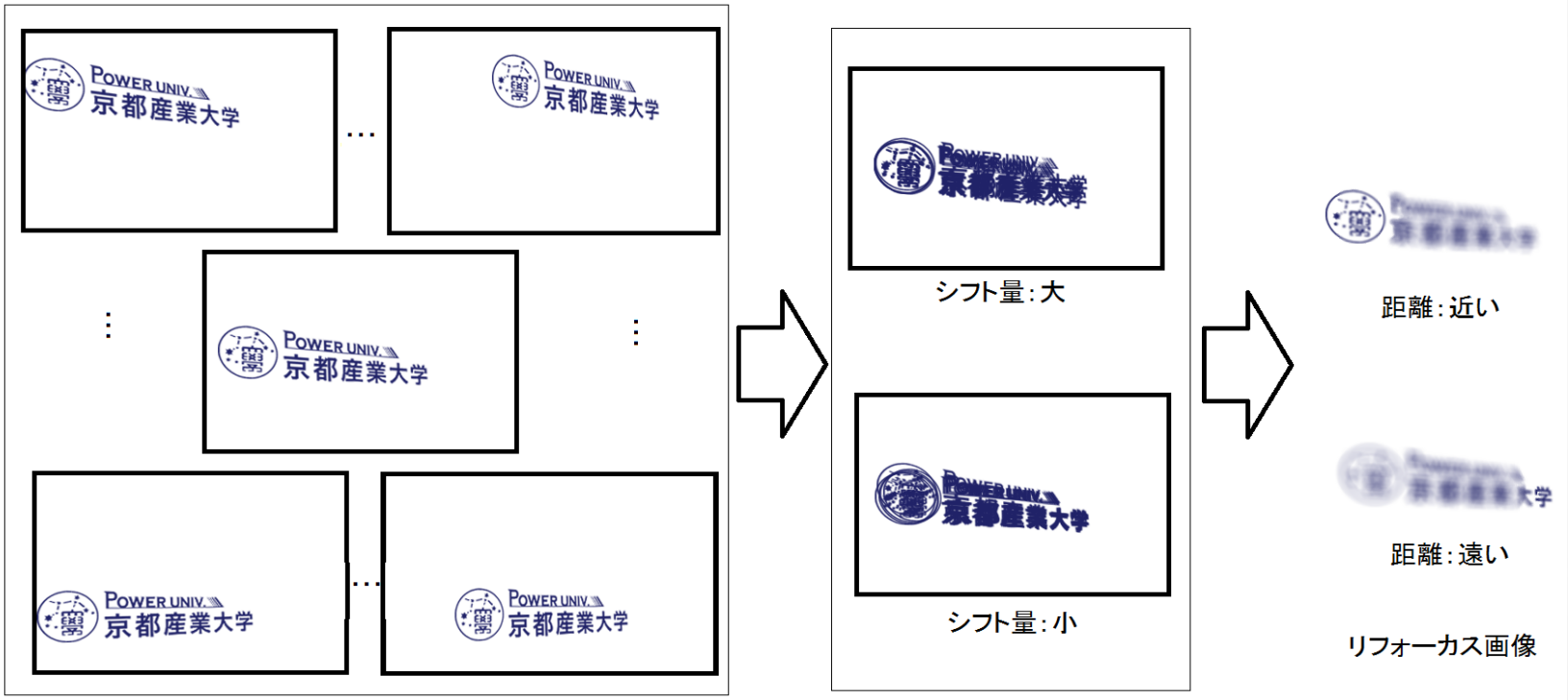
- GPU, CUDAの概要
- GPUの性能を最大化するプログムラム手法の提案
- 提案手法の画像処理への応用

実験

- 実際にGPUによる並列処理を画像処理に使用する実験を行う。

- リフォーカス画像の生成をはじめ、3種類の画像処理を行った。

リフォーカス画像の生成

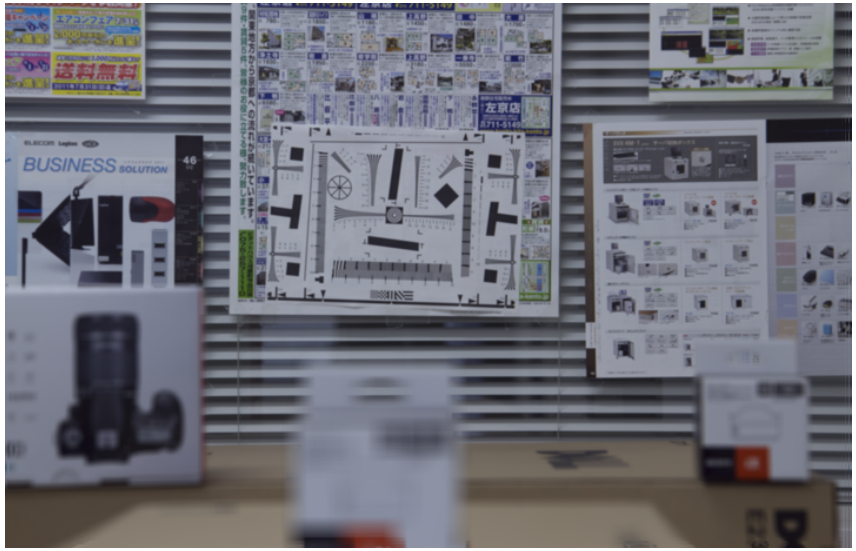


実験方法

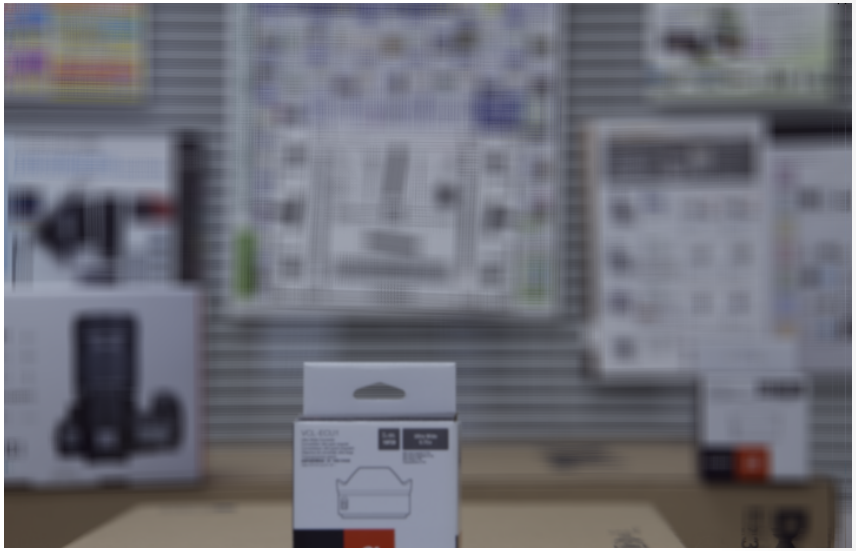
- 視差を有する720×480画素の7×7枚の画像を使用する。
- ピント位置の異なる30枚の画像を生成する時間を比較した。
- GPUではグローバルメモリのみ使用の場合と、シェアードメモリ併用の場合の2種類の処理を行う。

実験に使用した画像

リフォーカスされた画像



ピントが遠いときの画像



ピントが近いときの画像

処理内容

- 入力画像を適量平行移動させる処理
- 変更した入力画像を加算平均させる処理

逐次処理

画素00

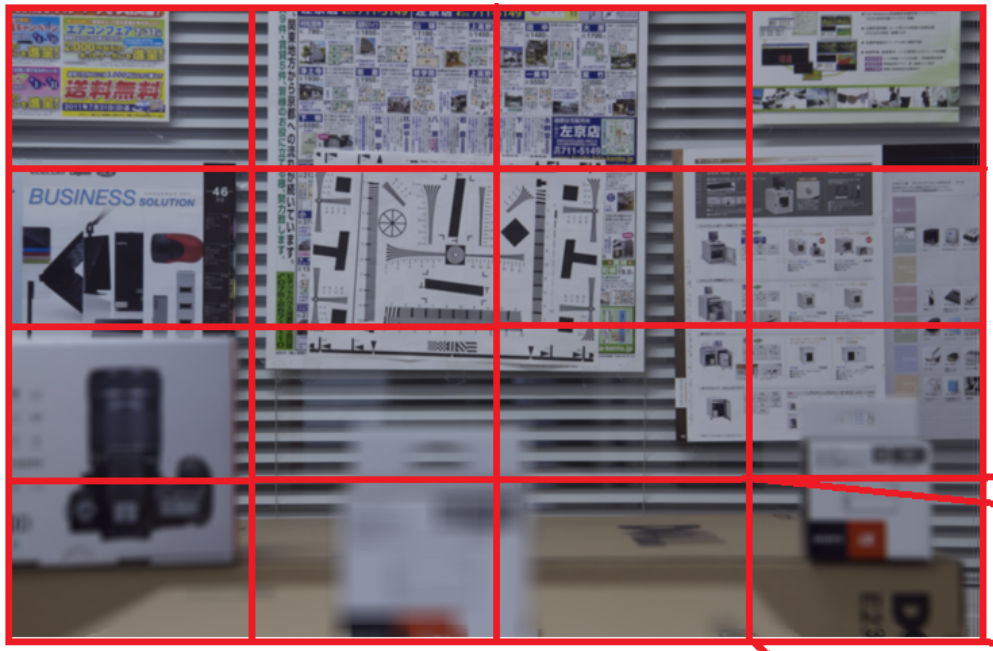
画素X0

画素0Y

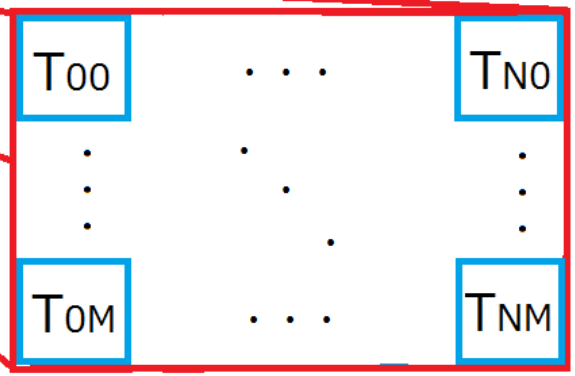
画素XY

画素00･･･画素X0,画素10･･･画素XYと1画素ずつ処理

並列処理



画像をブロックで分割



画素をスレッドに置き換えて処理

実験環境

- OS: Windows 7 64bit
- CPU : Intel(R) Xeon(R) CPU E5607 2.27GHz
- GPU : NVIDIA Tesla C2075 1.15GHz
- 開発ツール ： Microsoft Visual Studio 2008
- 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数 : 448基

実験結果

スレッド数	並列処理		逐次処理
	グローバルメモリのみ	シェアードメモリ使用	
4×4	1.057	0.976	40.289
8×8	0.712	0.653	
16×16	0.543	0.505	

逐次処理の80倍

ライトフィールドレンダリングの処理時間[sec]

その他の実験

- ラプラシアンフィルタによる画像のエッジの検出
- 動的計画法による画像のリサイズ

ラプラシアンフィルタの処理時間

スレッド数	並列処理		逐次処理
	グローバルメモリのみ	シェアードメモリ使用	
4×4	0.227	0.030	0.987
8×8	0.194	0.034	
16×16	0.187	0.032	

逐次処理の30倍

ラプラシアンフィルタの処理時間[ms]

動的計画法による画像のリサイズの処理時間

スレッド数	並列処理		逐次処理
	グローバルメモリのみ	シェアードメモリ使用	
4×4	0.0163	0.0127	1.3307
8×8	0.0159	0.0126	
16×16	0.0154	0.0126	

逐次処理の90倍

動的計画法による画像のリサイズの処理時間[ms]

GPUを画像処理に利用した場合の考察

- GPUの処理はCPUの処理に比べ、30～80倍の処理速度を実現した。
- 速度上昇の度合いに差が出た。→メモリアクセス回数の違いが原因か
- シェアードメモリを経由した画素のデータを使うことにより、より高速化された。
- スレッド数が増えるにつれて速度上昇の割合が減少→並列スローダウンが原因か

まとめ

- 処理の内容によって差は出るが、並列処理で行うことにより画像処理を高速化させることが出来た。
- プロセッサを単に多数使用するだけでは十分な高速化は達成できない。
- より高速に処理するためにメモリアクセス方法などの改善を行うことが重要。