特別研究報告書

汎用グラフィックスカードを用いた数値計算

指導教官　　富田 眞治 教授

京都大学工学部情報学科

篠本 雄基

平成16年2月2日

汎用グラフィックスカードを用いた数値計算

篠本 雄基

内容梗概

ゲーム等の表現技法としての 3D CG (Computer Graphics) の使用頻度の高まりから，汎用グラフィックスカードの処理能力は近年飛躍的に増大している. グラフィックスカードに搭載されるグラフィックスハードウェア（Graphics Processing Unit，以下 GPU と呼ぶ) は，従来固定された CG 処理を高速に実行する機能しか持っていなかった．近年の GPU は，より複雑な CG 処理を行うために，グラフィックスパイプラインの一部を，自由にプログラムしたパイプラインに置き換えることが可能となっている．また計算精度も向上しており，32 ビット単精度浮動小数点演算をサポートしている．このように，近年 GPU は複雑な計算を高い精度で行う能力を備えつつある．物理シミュレーションにおいて，これまでシミュレーションは CPU が全て行い，GPU は結果の可視化のみを行ってきた．従来全て CPU で行われていた計算の一部を，GPU が行うようにすれば，計算機全体としての計算能力が向上し，シミュレーション速度の向上が期待できる．このような背景に基づき，GPU を計算分野に適用することを考える．

GPU は，ポリゴンやテクスチャを，パイプライン方式を用いて処理し，レンダリングを行う．GPU が行う処理は，ベクトル計算機に搭載されているベクトルプロセッサに類似している．GPU を用いた数値計算では，CPU で配列データをテクスチャに変換し，GPU のパイプラインのプログラム機能を用いて，テクスチャのテクセル値を読み出し，操作を加えてポリゴンに貼り付けることで数値計算を行う．計算結果は画像として出力される．出力される画像をテクスチャにコピーすることで，配列要素を更新する．また，内容が更新されたテクスチャを，再び入力データとして用いることで，反復計算を行うことができる．GPU が持つ CG 処理機能を利用することにより，複雑な計算に対応することができる．

本研究では，実用的な数値計算のセットである L.L.N.L (Lawrence Livermore National Laboratory) fortran kernels に現れる代表的な反復パターンを抽出し，実装方法を検討した．実装方法の検討を通じて，様々な数値計算を GPU の機能を用いて実装する手法を考案し，現在の GPU の機能で，実装することが困難な計算の存在について調査した．次に，GPU の処理に適した計算として，2 次

元ポアソン方程式のヤコビ法による反復解法を GPU を用いて実装し，CPU を用いて反復解法を実装した場合との実行速度を比較した．

L.L.N.L. fortran kernels は 24 個のカーネルからなっている．実装方法を考案できなかったカーネルは 1 個のみであった．GPU が直接テクスチャに計算結果を格納する機能を持たないことが，1 個のカーネルの実装が困難な原因である．L.L.N.L. fortran kernels の反復パターンを分析した結果，GPU はループの各ステップに並列性がある計算に適しており，逐次計算および条件分岐を多用した計算には適していないことがわかった．2 次元ポアソン方程式のヤコビ法による反復解法は，収束判定を除いた部分に関しては，適切な分割数を選択すれば，32 ビット単精度浮動小数点の精度で，GPU での実装が CPU での実装より，約 10%高速であるという計測結果になった．

本研究では，様々な数値計算を，汎用グラフィックスカードに搭載される GPU を用いて実装するための汎用的な手法を考案し，GPU がベクトルプロセッサとして十分な機能を持っていることを示した．次に，GPU に適した処理に関しては，CPU よりも高速であることを計測結果によって示し，GPU を用いた数値計算の有用性を確認した．さらに，現在の GPU で実行が困難である計算の存在を指摘した．

Numerical Computations on A Commodity Graphics Hardware

Yuki SHINOMOTO

Abstract

Recently, a computational performance of a commodity graphics hardware has made a rapid progress with an increasing use of 3D CG (Computer Graphics) as an expression of video games. Graphics hardwares used to have only fixed CG processing functions. As a recent functional progress, graphics processing units (GPUs) on graphics hardwares can programaticaly reconfigure the graphics pipeline. With regards to a precision of computations, GPUs can use 32-bit floating point. In scientific simulations, CPU computes reults of simulations. GPUs don't involve with simulations and only visualize the results. if CPU and GPU work together and GPU is used as a vector processor, simulations can be faster. In this paper, we apply GPUs to numerical computations.

GPUs are a kind of vertor processors. GPUs have the vertex processor, the fragment processor, and other non-programmable hardware units which are linked through data links. When GPUs deal with arrays, they use textures which contain array elements as texels. The programmable fragment processor samples data of textures and process them. Results of computations are preserved by copying the results to textures or readback the frame buffer/offscreen buffer into CPU memory. Note that GPUs don't update array elements directly. By updating texture with the results, GPU can do iteration.

In this research, we analyzed L.L.N.L (Lawrence Livermore National Laboratory) fortran kernels which are sets of practical numerical computations. We picked characteristics of L.L.N.L fortran kernels and considerd how to implement them and what kind of computation is difficult to implement with GPUs. Also we implemented a 2-D Poisson solver with a Jacobian method on NVIDIA GeForce FX 5900 Ultra and on Intel Pentium 4 each and compared computational time of CPU and GPU.

L.L.N.L. fortran kernels have 24 kernels. We worked out methods of implementing them execpt for one kernel and found that gpus' inability to update array elements directly limits kinds of computations. Also we found that GPUs

are good at parallel computations and poor at one-by-one or branching computations by analyzing patterns of L.L.N.L. fortran kernels. A 2-D Poisson solver with a Jacobian method using GPU was about 10 percent faster than using CPU at precision of 32 bit floating point when we omittied convergence check and selected a proper number of divided grids. We believe that in real-time simulations GPU should display results of simulations with real-time frequency while CPU displays precise ones. And GPU may iterate prefixed times or check the convergence every prefixed steps other than check theconvergence every step.

In this research, we worked out general methods of implementing a variety of numerical computations and pointed out computations which is difficult to implement with GPUs. Also we showed the usefulness of numerical computations on a commodity graphics hardware through implementing A 2-D Poisson solver. A current problem is that 32-bit floating point operations are very expensive in GPUs and that there are some limitations in computing on GPU. Another problem is that CPU always sents commands to GPU and GPU can't work by itself. When these problems are resolved, GPU will coorporate with CPU as a powerful vector processor.

汎用グラフィックスカードを用いた数値計算

目次

第1章　はじめに

ゲーム等の表現技法としての 3D CG (Computer Graphics) の使用頻度の高まりから，汎用グラフィックスカードの処理能力は近年飛躍的に増大している．機能面に注目すると，グラフィックスカードに搭載されるグラフィックスハードウェア（Graphics Processing Unit，以下 GPU と呼ぶ) が，従来固定されたごく少数のレンダリングパイプラインしか持っていなかったのに対して，CG 表現の幅を広げるため，グラフィックパイプラインの一部を自由にプログラムしたパイプラインに置き換えることが可能となっている．また計算精度も向上しており，32 ビット単精度浮動小数点演算をサポートするようになっている．このように，近年 GPU は複雑な計算を高い精度で行う能力を備えつつある．物理シミュレーションにおいて，これまでシミュレーションは CPU で全て行われてきた．GPU はシミュレーション結果の可視化にのみ使用され，シミュレーション中に GPU は完全にアイドルになっていた．シミュレーションにおいて CPU の行う計算の一部を，GPU が行うようにすれば，計算機全体としての計算能力が向上し，シミュレーション速度の向上が期待できる．このような背景にもとづき，GPU を計算分野に適用することを考える．GPU がもっとも得意とする処理内容である CG 処理として数値計算を行わせるのであれば，GPU の持つ CG 処理専用のハードウェア機能を利用でき，高速に処理を実行することができると考えられる．本稿では，グラフィックスカードの構成についてまとめ，次に GPU を用いて数値計算を行う手法について説明し，さらに実用的な数値計算のセットである L.L.N.L (Lawrence Livermore National Laboratory) fortran kernels の実装方法を検討したのち，2 次元ポアソン方程式の反復解法を GPU を用いて実装する．

第2章　背景

2.1　グラフィックスカードの概要

2.1.1　用語の定義

幾何学的データ　幾何学的データは頂点群と，それが記述するプリミティブの形式（頂点，線，ポリゴン）から構成される．頂点データには座標だけでなく，法線ベクトル，テクスチャ座標，RGBA カラー，材質特性も含まれている．

レンダリング (Rendering)　幾何学的データを，画像に変換する処理である．

テクスチャ (Texture)　テクスチャとは，物体の表面の質感を表現するためにポリゴンに貼り付ける画像である。テクスチャ要素は，テクセルとよばれる．

頂点プロセッサ (Vertex Processor)　形や色などを定義したモデルと光源を３次元のステージ上に配置し，視点から見たそれらの形状や位置関係，光源が及ぼす明暗や陰影などの効果などを計算し，立体的な画像を描画していく．近年，ポリゴンの位置や物体に対する光の加減を計算する頂点シェーディングを，モデリングデータの頂点情報をモデリングデータを変更することなく，プログラマブルに行うことが可能になっている．

トライアングルセットアップ (Triangle Setup Engine)　頂点プロセッサでのジオメトリ演算やライティングにより作成された点，線，ポリゴン，画像を画素の集まりに分解し，ビットマップデータを作成する．画素に分解する操作をラスタライズという．分解された画素をフラグメントと呼ぶ．フラグメントは，単一のピクセルに対応する．

フラグメントプロセッサ (Fragment Processor)　内部にテクスチャ操作を行うテクスチャユニットを持ち，ラスタライズされたポリゴンや背景などに対して，テクスチャを貼り付ける．近年，モデルの表面を処理するフラグメントシェーディングをプログラマブルに行うことが可能になっている．

Z バッファ (Z Buffer)　各ピクセルからの距離を保存する．Z バッファの値が大きいピクセルは，小さな値のピクセルに上書きされる．

フレームバッファ(Frame Buffer)　これらの処理結果を記憶しておく．

RAMDAC　フレームバッファのデジタルデータを，ディスプレイに出力できるようにアナログデータに変換する．

ビデオメモリ (Video Memory)　レンダリングパイプラインが処理する前のピクセルデータの一時的な展開場所として，また前述の Z バッファやフレームバッファとして使用される．また，テクスチャデータを格納しておく場所でもある．

グラフィックス API (Graphics API)　グラフィックスカードにアクセスするために呼び出す関数群であり，プログラマはグラフィックス API を用いることで，容易にグラフィックスカードの機能を利用することができる．Windows 環境で用いられる DirectX，OS に依存しない OpenGL などがある．

2.1.2 汎用グラフィックスカード

汎用グラフィックスカードとは，ディスプレイ画面への画像表示機能を持った拡張カードのうち，パーソナルコンピュータに搭載される比較的安価な物を指す．市販されている汎用グラフィックスカードに搭載されている GPU の例として，NVIDIA 社の GeForce，ATI 社の Radeon，Matrox 社の Parhelia などが挙げられる．GPU には，頂点プロセッサとフラグメントプロセッサに加えて，その他の非プログラマブルなハードウェアユニットが搭載されている．プロセッサ，非プログラマブルなユニットおよびアプリケーションは，すべてデータフローによってリンクされる．図 1 に GPU のグラフィックスパイプラインを示す．

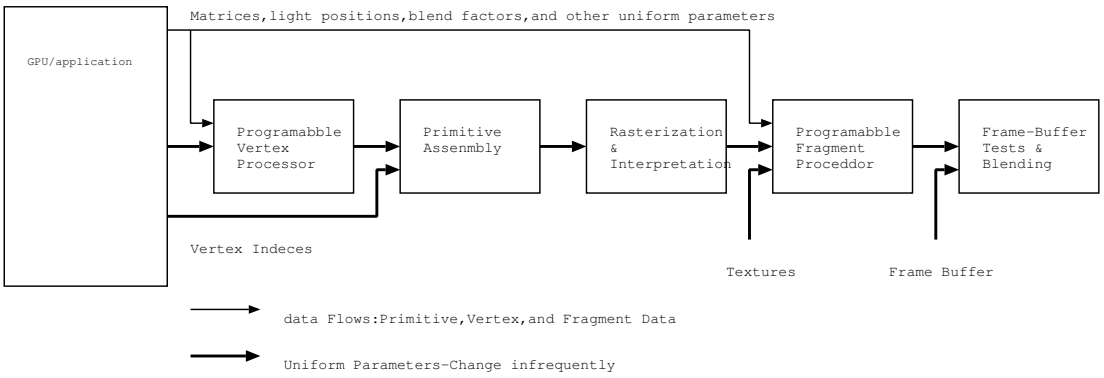

図 1: グラフィックスパイプライン

GPU は，グラフィックス API の機能拡張に対応する形で，機能を向上させてきた．DirectX を例に取ると，DirectX 7 以前の GPU は，固定された機能しか持っておらず，主要な CG 処理は,Hardware T&L (Hardware Transformation and Lighting) とよばれる座標変換と，ライティング処理であった．DirectX 8 対応の GPU において，より複雑な CG 処理を行うために，頂点プロセッサとフラグメントプロセッサに，プログラマブルパイプラインの概念が導入された．プログラマブルパイプラインは，プログラムすることでパイプラインの内容を変更することができる．DirectX 9 対応の GPU は，プログラマブルパイプラインの機能が強化され，より汎用的な計算を行えるようになった．

2.1.3 プログラマブルシェーダ

GPU の頂点プロセッサをプログラムする言語を頂点プログラム，フラグメントプロセッサをプログラムする言語をフラグメントプログラムと呼び，2 つ

を合わせてシェーダ言語と呼ぶ．広義のシェーダ言語には，非リアルタイム CG 作成用のシェーダ言語も含まれるが，本稿では，GPU 用シェーダ言語を指すものとする．GPU の種類やグラフィックス API によって使用できるシェーダ言語が異なる．頂点プログラムの例として DirectX の VS2.0 や OpenGL の GL_ARB_vertex_program，フラグメントプログラムの例として DirectX の PS2.0 や OpenGL の GL_ARB_fragment_program などがあり，加えて GPU メーカーが独自に制定した言語がある．それぞれ使用できる命令セットや命令数がことなる．シェーダ言語は，GPU の性能の向上に合わせて機能が向上している．NVIDIA 社の OpenGL 拡張である NV_vertex_program2 および NV_fragment_program の仕様を表 1 に示す．両プログラムは，現在最も豊富な機能を持つシェーダ言語である．本稿では，以降シェーダ言語として NV_vertex_program2 および NV_fragment_program を想定する．

	NV_vertex_program2	NV_fragment_program
入力	頂点 テクスチャ座標 カラー	テクスチャ カラー
	法線 フォグ座標 等	テクスチャ座標 ウィンドウ位置座標
出力	位置座標 カラー フォグ座標 等	カラー 深度
定数レジスタ	255 個	512 個
最大静的命令数	255 個	1024 個
最大実行命令数	65536 個	1024 個

表 1: シェーダ言語の仕様

最大静的命令数とは，1 回のレンダリング内で設定できるプログラムの命令数であり，最大実行命令数とは，レンダリング中に分岐先を展開した場合の総実行命令数である．両プログラム間には，頂点プロセッサおよびフラグメントプロセッサの処理内容を反映した相違点が存在する．

- 頂点プログラムは頂点ごとに適用され，フラグメントプログラムはフラグメント毎に適用される．
- 動的な分岐命令は，頂点プログラムのみが持つ．
- テクスチャにアクセスする命令は，フラグメントプログラムのみが持つ．
- フラグメントプログラムは，フラグメントを破棄する命令を持つ．

2.1.4 上位レベルシェーダ言語

頂点プログラムおよびフラグメントプログラムは，アセンブリ言語の形式をとっており，従来プログラミングが複雑であった．近年，Cg (C for Graphics)，HLSL (High Level Shader Language) などの GPU プログラミング用の高級言語が NVIDIA 社，microsoft 社によって開発されている [2]．これらの言語は，CPU における高級言語がアセンブリコードにコンパイルされるのと同様に，専用のコンパイラを用いて前述のシェーダ言語にコンパイルされる．コンパイルされるシェーダ言語の種類は，オプションで指定できる．そのため，使用するグラフィックスカードの機能やグラフィック API に合わせたシェーダ言語を選択できる．

第 3 章　GPU を用いた数値計算の手法

GPU の頂点プロセッサおよびフラグメントプロセッサが行う処理は，以下のような特徴を持つ．

- 各パイプラインは，命令のオペランドと結果の格納先が，GPU 内のレジスタである．
- ピクセルの色情報である RGB アルファ(赤，緑，青，不透明度) は，1 ベクトルとして 1 命令でまとめて演算できる．
- RGB アルファの値を入れ替えて，新しいベクトルを作ることができる．例えば，RRRR や BGG といったベクトルを作ることができる．なおこの操作は計算コストがかからない [6]．
- パイプラインにより，複数の頂点およびピクセルが同時に処理される．
- 全ての頂点およびピクセルに対して，同一の計算を施す．
- 異なる頂点およびピクセルの演算は，相互に干渉できない．

これらの特徴は，レジスタ-レジスタ型ベクトル計算機のベクトルプロセッサと類似しており，GPU はベクトルプロセッサの一種であるといえる．GPU がベクトルプロセッサと異なる点として，以下があげられる．

- 頂点プロセッサ，フラグメントプロセッサという 2 つのベクトルプロセッサを持つ．
- あるパイプラインの計算結果を，他のパイプラインの入力にする技法をチェイニングと呼ぶ．GPU は自動的に頂点プロセッサの計算結果が，フラグメ

ントプロセッサの入力にチェイニングされるが，チェイニングするパイプラインを任意に選択することはできない．

- 演算結果はフレームバッファに画像として出力される．
- 出力画像から読み出せる情報は，ピクセルの RGB アルファの値のみであり，プログラムの計算結果の出力に制限がある．

3.1 データ入力

GPU による数値計算においては，解析の対象である計算データを，CPU で GPU が本来扱うデータフォーマットに基づいたデータに変換して GPU のビデオメモリに転送し，それらがグラフィックスパイプライン上を流れることになる．頂点プログラムにおいては各頂点が持つ頂点座標や色といった値にアクセスし，フラグメントプログラムにおいては各ピクセルが持つテクスチャ座標や色といった値にアクセスする．また定数レジスタに頂点やピクセルと関係のないデータを割り当て，アクセスすることもできる．行列，配列といった大規模な数値データは，テクスチャに変換する．本研究では，テクスチャに対して操作を加えることで計算を行う．これは，テクスチャが以下のような特徴を持つためである．

- レンダリングをする際に入力とすることができ，レンダリングの出力先をテクスチャとすることができる．
- 単純な符合なしフォーマットによる RGB アルファの色情報だけではなく，符合付きフォーマット，IEEE 浮動小数点フォーマットといった様々な形式でデータを保持することができる．
- 保持される数値データの解釈および使用方法は，プログラマブルパイプラインにおいては自由にカスタマイズすることができる．
- 複数のピクセルに同じテクセルを対応させることで，テクスチャサイズより大きなポリゴンに，テクスチャを引き伸ばして貼り付けることができる．

3.2 配列要素の参照

配列データを格納したテクスチャは，ポリゴンの頂点座標にテクスチャ座標を対応させることでポリゴンに貼り付けられる．配列要素の値は，テクスチャ座標を指定してテクセルの値を読み出すことで参照する．数値計算で頻繁に登場する，全ての配列要素に対しての計算は，テクスチャと同サイズのポリゴン

にテクスチャを貼り付け，テクスチャ座標を配列の添え字に一致させた場合，全てのテクセルに対しての計算と同等である．

3.3 結果の出力対象

通常のレンダリングは，フレームバッファをターゲットとして行われる．フレームバッファはディスプレイ環境に依存するため，GPU 内部での計算精度を反映しない．例えば RGB アルファの値は，[0.0,1.0] の範囲に切り取られる．近年，ディスプレイに表示されない，ピクセルバッファとよばれるオフスクリーンバッファを，ビデオメモリ内に作成する機能を GPU がサポートするようになった [7]．ピクセルバッファは，フレームバッファと同様に描画対象とすることができるが，ウインドウサイズおよびデータフォーマットはディスプレイ環境に依存しない．ピクセルは, [0,1] の範囲を超えて，32 ビット IEEE 浮動小数点の範囲の値を持つことができる．ピクセルバッファをターゲットとしてレンダリングすることで，計算精度を保ったまま演算を繰り返すことが可能になる．したがって，数値計算はピクセルバッファとターゲットとして行うことになる．

3.4 データの保持

計算結果は，各ピクセルの色情報としてピクセルバッファに出力される．ピクセルバッファの内容を CPU のメモリに書き戻すことで，計算結果を数値として取り出せる．また反復計算を GPU 内で行う場合などは，ビデオメモリ内に 1 つ前の反復における計算結果を保持する必要がある．この場合，2 通りの方法がある．ピクセルバッファへの出力内容を既存のテクスチャにコピーする方法と，ピクセルバッファへの出力内容そのものをテクスチャにする方法である．後者の方法は，近年導入されたレンダーテクスチャとよばれる機能を用いて実現される [7]．レンダーテクスチャは，CPU のメモリからテクスチャデータを転送せずに，GPU 内でテクスチャを生成できるため，AGP バスの帯域消費を節約できる．配列要素への値の代入は，ピクセルバッファへの出力結果で，テクスチャを更新することに対応する．

3.5 プログラミング

頂点プログラムおよびフラグメントプログラムを用いて行う．両プログラムは，四則演算を初めとして，行列演算，三角関数，平方根，対数，指数関数と

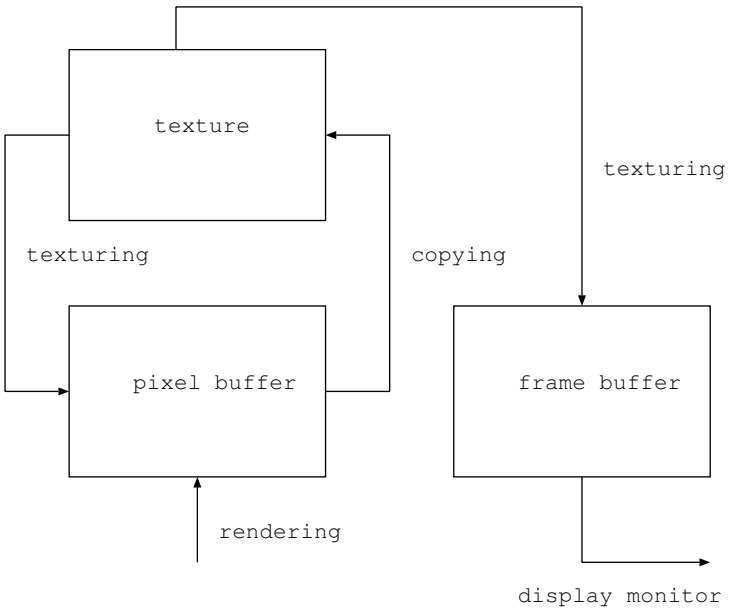

図 2: 反復計算

いった豊富な数学関数を持っている．テクスチャのテクセル値は，現在フラグメントプログラムにおいてのみ参照できるため，数値計算はフラグメントプログラムが中心となる．

3.6 条件分岐

現在，動的な条件分岐は頂点プログラムでのみサポートされている．動的な条件分岐とは，条件にしたがってプログラムの流れを変え，分岐先の命令のみ実行することである．フラグメントプログラムで条件分岐を行う場合は，条件にしたがって 0,1,-1 の値を返す条件セット命令を用いる．分岐先の両方の命令を実行したのち，条件を満たさない分岐先の命令を条件セット命令でマスクすることで条件分岐を実現する．この方法で条件分岐を行うと，実質命令数が 2 倍になるため，実行速度が低下する．また，分岐先を展開できない場合は分岐できない．

3.7 高速化手法

演算のベクトル化 GPU はグラフィックス処理の目的に特化されており，ピクセルの色情報である RGB アルファ 4 値の計算をベクトル演算として同時に行う．このため，演算をベクトル化できる場合は，RGB アルファの全ての

要素に値を格納することで，高速に処理できると考えられる．

処理の分担　頂点プログラムは全頂点に対して実行され，フラグメントプログラムは全ピクセルに対して実行される．一般的にフラグメントプログラムは頂点プログラムより実行回数が多いため，両プログラムで実行可能な処理は，頂点プログラムで行うことにより，高速化を図られると考えられる．

ディスプレイリスト　描画コマンドは，全て CPU 側から送信され，GPU の画像処理専用ハードウェアで実行される．反復計算では，CPU から同じ描画コマンドが何度も送信されることになり，オーバーヘッドになる．描画コマンドをディスプレイリストとよばれる命令リストにまとめてビデオメモリに記憶し，同じ描画コマンドを送信する代わりに，ディスプレイリストからコマンドをまとめて呼び出すことで，CPU-GPU 間の通信のオーバーヘッドが削減され，高速化が図られる．

テクスチャサイズの圧縮 CPU 側のメモリから GPU 側のビデオメモリにデータを転送する際，データは AGP バスを通る．AGP バスは最大で 2Gbps の帯域しか持っておらず，サイズの大きいテクスチャを転送する際にボトルネックとなる可能性がある．また GPU においても，サイズの大きなテクスチャはメモリ帯域を消費する．そのため，データ転送量をできるだけ少なくする必要がある．数値計算においては，疎行列とよばれる, 要素の値がほとんど 0 である行列を扱うことがある．疎行列をそのままテクスチャにすると，ほとんどのテクセル値に 0 が入ることになり，データに無駄が生じる．値が 0 でない部分だけをテクスチャとして GPU に送り，テクスチャ座標を調節してポリゴンに貼り付ければ，少ないデータ量で元の疎行列を再現できる．

第 4 章　実装

本研究では，L.L.N.L fortran kernels の GPU を用いた実装方法について検討し，次に 2 次元ポアソン方程式の差分方程式の，ヤコビ法による反復解法を GPU を用いて実装した．

4.1 実装環境

OS は Windows XP，グラフィックス API は OpenGL 1.4，シェーダ言語は NV_vertex_program2 および NV_framgent_program をそれぞれ用いた．GPU は，NVIDIA 社製 GeForce FX 5900 Ultra を用い，CPU は，Intel 社製 Pentium4 3.00GHz を用いた．表 2 に GeForce FX 5900 Ultra の性能を示す．

コアクロック	450MHz
メモリ転送レート	850MHz
ビデオメモリインターフェイス幅	256bit
メモリ帯域	27.2GB/sec
標準搭載メモリ	256MB
ジオメトリ性能	338.0MVertices/sec
フィルレート	3.60Gtexels/sec
ピクセル/クロック	8(4)
計算精度	IEEE32 ビット単精度浮動小数点
API サポート	OpenGL1.4，DirectX9.0

表 2: GeForce FX 5900 Ultra の性能

GeForce FX 5900 Ultra は，計算精度によってフラグメントプロセッサの 1 クロックあたりの処理ピクセル数が変動する．IEEE32 ビット単精度浮動小数点フォーマットを用いる場合は，4 ピクセル/クロックとなる．ATI 社の Radeon では，頂点プロセッサは 32 ビットの精度で計算し，フラグメントプロセッサは，浮動小数点の計算精度を 24 ビットに落とすことで，8 ピクセル/クロックで処理する仕様となっている．

4.2 L.L.N.L fortran kernels の実装

L.L.N.L fortran kernels は，物理シミュレーションなどの数値計算で行われる基本的な演算のセットであり，過去にベクトル型計算機のベンチマークとして用いられた．L.L.N.L fortran kernels を GPU で実行できれば，幅広い数値計算を GPU で実行できると考えられる．本研究では，24 個あるカーネルの中に現れ

る代表的な反復パターンを抽出し，GPU を用いた実装方法について検討した．

4.2.1 kernel1 HYDRO FRAGMENT

```
DO 1 k = 1,n
   1 X(k)= Q + Y(k) * (R * ZX(k+10) + T * ZX(k+11))
```

流体力学における計算の断片である．ループの各ステップ間の依存性がなく，並列性があるため，高さ 1，幅が X の配列長である矩形ポリゴンにテクスチャを貼り付けることで，X の全要素が計算できる．GPU がもっとも得意とする計算である．配列 Y，ZX はテクスチャとして GPU に送る．Q は全ての計算で加算されるため，各ピクセルで個別に計算する必要がない．このような場合は，頂点プログラムで加算し，フラグメントプログラムで残りの計算を行うことで，高速化を図る．配列の複数の要素を参照することは，1 つのテクスチャの複数のテクセル値を読み出すことに対応する．k+10，k+11 は各ピクセルのテクスチャ座標のオフセットと考える．オフセットは頂点プログラムで計算し，複数のテクスチャ座標としてフラグメントプログラムに送る．テクセル値の RGB アルファ全てに配列要素を割りふり，4 つの要素をベクトルとしてまとめ，ベクトル演算を行うことで，さらに高速化できると考えられる．

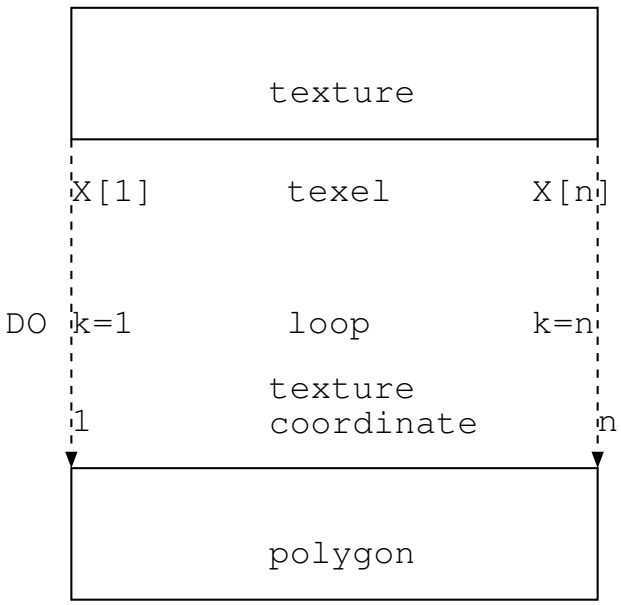

図 3: ループとテクスチャ座標の対応

4.2.2 kernel2 ICCG EXCERPT (INCOMPLETE CHOLESKEY-CONJUGATE GRADIENT)

ICCG 法 (前処理付き共役勾配法) の断片である．この計算の実装で問題となるのは以下の箇所である．

```
i= IPNTP+1
 DO 2 k= IPNT+2,IPNTP,2
      i= i+1
   2   X(i)= X(k) - V(k) * X(k-1) - V(k+1) * X(k+1)
```

ループの各ステップに並列性はあるが，右辺の参照する配列の添え字 i と，左辺の代入される配列の添え字 k の増加する幅が異なる．1 つ目の方法では，配列の添え字 i に対応する k の値を格納した配列をテクスチャとして用意し，k の代わりに，用意したテクスチャのテクセル値を右辺の添え字とする．2 つ目の方法では，各ピクセルのテクスチャ座標は，ポリゴンの各頂点に対応させたテクスチャ座標を基準として，自動的に線形補間されるという性質を利用する，ループが終了する時点での i の値を CPU の計算で予め用意しておく．テクスチャ座標を 2 種類用意し，i=IPNT+1 と k=IPNT+2，i の最終値と k=IPNTP をそれぞれ同じポリゴン頂点に対応させれば，i にあたるテクスチャ座標が 1 増加する度に，k にあたるテクスチャ座標は 2 増加する．前者の方法は CPU 側でテクスチャを用意しなければならないため，後者の方法がより高速だと考えられる．このように，テクスチャサイズより小さなポリゴンにテクスチャを貼り付けることで，ループの増加幅が 1 以外の計算に対応できる．

4.2.3 kernel3 INNNER PRODUCT

```
DO 3 k= 1,n
   3       Q= Q + Z(k) * X(k)
```

内積計算は，パイプライン毎の計算結果を足し合わせる作業がオーバーヘッドになるため，ベクトル型計算機が不得意とする計算である．GPU では，まず内積を取る 2 つの行列を 2 枚のテクスチャとする．GPU はベクトル長 4 までの内積をサポートしており，高速に演算できるため，テクセルの RGB アルファ全てに値を格納できるようテクスチャの生成を工夫する．まず 2 枚のテクスチャを用いてテクセル毎の内積を計算し，ピクセルバッファに出力する．次にピクセルバッファの内容をテクスチャにコピーし，2 回目以降のレンダリングで全て

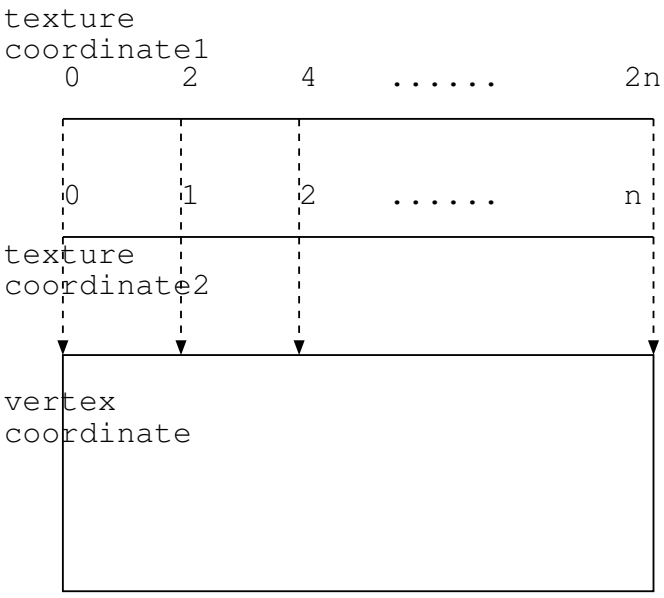

図 4: ループの増加幅の調節

のテクセルの値を加算する．加算する部分では，一度に加算するテクセル数を決定し，テクスチャを貼り付ける矩形ポリゴンの幅を，元の幅/加算する要素数に縮小する．そしてポリゴンにテクスチャ座標をずらして同一のテクスチャを何枚か重ね合わせて値を加算する．テクスチャを重ね合わせる枚数=一度に加算する要素数となっている．この計算結果でテクスチャの内容を更新する．これを繰り返すことで，配列の各要素が加算されていき，最終的には幅 1 高さ 1 のテクスチャに内積の計算結果が格納される．1 つのポリゴン頂点に設定できるテクスチャ座標の数が多いほど，一回の反復でまとめられる配列の要素数が多くなる．

4.2.4 kernel5 TRI-DIAGONAL ELIMINATION,BELOW DIAGONAL

```
DO 5 i = 2,n
   5    X(i)= Z(i) * (Y(i) - X(i-1))
```

三重対角行列除去の断片である．ループの各ステップにおいて，1 つ前のステップの計算結果を使用するため，各ステップの計算をベクトル化して並列に行うことができない．GPU で行うには不向きな計算である．1 ピクセルの RGB アルファ全てに結果を格納すれば，一回のレンダリングでループを 4 ステップ

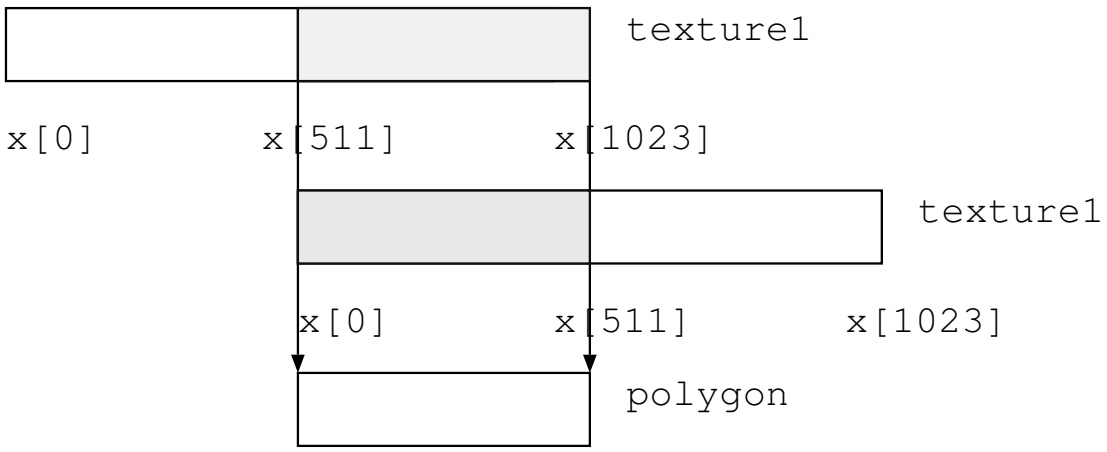

図 5: 配列要素の足し合わせ

先まで処理できる．実装としては，まず配列 X を格納するテクスチャを用意し矩形ポリゴンに貼り付ける．計算結果が格納される X(1) 以外の部分をクリッピングして描画を行わないようにしておき，X(1) の値を計算する．ピクセルバッファの内容のうち X(1) にあたる部分を配列 X のテクスチャに上書きする．以降順に X(i) の値を計算し，当該部分のテクスチャを上書きすることを繰り返す．

4.2.5 kernel8 A.D.I. INTEGRATION

偏微分方程式の解を求める際に用いられる．計算として特徴的なのは以下の部分である．

```
      nl1 = 1  nl2 = 2
U1(kx,ky,nl2)=U1(kx,ky,nl1)+A11*DU1(ky)+A12*DU2(ky)+A13*DU3(ky)
 +SIG*(U1(kx+1,ky,nl1) -fw*U1(kx,ky,nl1) +U1(kx-1,ky,nl1))
U2(kx,ky,nl2)=U2(kx,ky,nl1) +A21*DU1(ky) +A22*DU2(ky)+A23*DU3(ky)
 +SIG*(U2(kx+1,ky,nl1) -fw*U2(kx,ky,nl1) +U2(kx-1,ky,nl1))
U3(kx,ky,nl2)=U3(kx,ky,nl1) +A31*DU1(ky) +A32*DU2(ky)+A33*DU3(ky)
 +SIG*(U3(kx+1,ky,nl1) -fw*U3(kx,ky,nl1) +U3(kx-1,ky,nl1))
```

3 次元配列を使用している．通常 3 次元配列は 3 次元テクスチャに格納するが，2 次元テクスチャは RGB アルファをテクセル値として持つため，3 次元の要素数が 4 を超えなければ，2 次元テクスチャで 3 次元配列を表現できる．RGB アルファが同じ計算をする場合は，ベクトル計算として高速に処理できるので，データの格納方法を工夫して，処理をベクトル化する．ここでは 3 次元配列 U1,U2,U3 の 3 次元の要素数が 2 であるので，3 次元配列を 2 つの 2 次元配列に分け，U1,U2,U3

を 1 つのテクスチャにまとめる．同様に，A11,A21,A31，A11,A21,A31 および A13,A23,A33 は，フラグメントプログラム内でそれぞれ 3 要素ベクトルとして定義する．U1,U2,U3 の計算に用いる配列の添え字が全て同じであるため，ベクトル計算として実行できる．

4.2.6 kernel10 DIFFERENCE PREDICTORS

差分予測を行うカーネルである．

```
DO 10  k= 1,n
      AR      =      CX(5,k)
      BR      = AR - PX(5,k)
      PX(5,k) = AR
      CR      = BR - PX(6,k)
      PX(6,k) = BR
```

以上の計算を繰り返して PX(13,k) まで計算する．ループの各ステップに並列性があるが，各ステップ内の計算は逐次に行う必要がある．1 ピクセルにつき，4 つの値しか出力できないため，値が更新された配列要素を 1 回のレンダリングで全て出力することはできない．このため，1 ステップの計算を分割し，テクスチャを上書きしながらレンダリングを繰り返す必要がある．その際，AR,CR,BR の値を記憶しなければ次の配列要素が計算できないため，配列 P の値とともにテクスチャに格納する．計算結果の値を 1 テクセル毎に 5 個以上保持したい場合は，命令を分割して複数回レンダリングすることになる．

4.2.7 kernel14 1-D PIC Particle In Cell

1 次元 PIC (Particle In Cell) 法の断片である．GPU での実装が難しいのが，以下の計算である．

```
DO  14    k= 1,n
      RH(IR(k)  )= RH(IR(k)  ) + fw - RX(k)
      RH(IR(k)+1)= RH(IR(k)+1) + RX(k)
```

参照する配列 RH の添え字が，配列 IR の値となっており，間接参照を行う．そのため配列 RH の更新する要素が，k とは無関係になる．配列 IR の値によっては，ループの各ステップに依存関係が生じるためベクトル化できない．GPU の処理では，現在のテクスチャ座標で配列 IR の要素を参照して新たなテクスチャ座標をとし，テクスチャRH の要素を参照することに対応する．各パイプラインは，自分が処理しているピクセル以外の他のパイプラインが処理している

ピクセルに干渉できないため，直接他の位置のピクセルに結果を出力できない．あらかじめ配列 IR を CPU のメモリにも保持しておく．ピクセルバッファに結果を出力し，IR(k) の値を CPU が読み出し，IR(k) の位置にあるテクセルを出力結果で置き換える．この操作を 1 反復ごとに行う．kernel5 と同じく，GPU で行うには不向きな計算である．

4.2.8 kernel15 CASUAL FORTRAN. DEVELOPMENT VERSION.

```
      DO 45  j = 2,NG
      DO 45  k = 2,NZ
             IF( j-NG) 31,30,30
30      VY(k,j)= 0.0d0
                GO TO 45
31           IF( VH(k,j+1) -VH(k,j)) 33,33,32
```

スカラー演算を順序付けるカーネルである．条件分岐を多用しており，また，条件によってループ内の計算を途中で中断する，いわゆる continue 命令が存在する．条件分岐を完全に展開できるため，フラグメントプログラムで実装できる．ループ内の処理の中断は，中断時点での計算結果をバッファに出力することで実現する．ループの各ステップは並列に実行でき，また各ステップの中断は他のステップの実行に影響しないため，continue 命令を実現できる．

4.2.9 kernel17 IMPLICIT,CONDITIONAL COMPUTATION

条件分岐があり，計算結果を格納する配列の位置とプログラム内のループの回数が，配列の値を元に動的に決定される．更新される配列要素の数と位置が事前に把握できず，ピクセルの色情報として何の値を出力するかを決定できない．そのため，現時点での実装方法は考案できていない．

4.2.10 kernel22 MATRIX*MATRIX PRODUCT

```
DO 21 k= 1,25
      DO 21 i= 1,25
      DO 21 j= 1,n
      PX(i,j)= PX(i,j) +VY(i,k) * CX(k,j)
```

行列の掛け算である．計算に並列性があり，高速に処理できると考えられる．特徴はループが三重になっていることである．一番外側の k のループをフラグメントプログラムで行う．このループは完全に展開できるため，フラグメントプログラムでも実行できる．i と j のループはテクスチャを矩形ポリゴンに貼り

付けて，ピクセルを計算することで実現される．シェーダプログラム内のループは，展開した際にプログラムの最大命令数を超えてはならないので，CPU で実行できるループが GPU では実行できない場合がある．

4.2.11 kernel24 FIND LOCATION OF FIRST MINIMUM IN ARRAY

```
DO 24  k= 2,n
    IF( X(k).LT.X(m))  m= k
```

配列要素の最小値の位置を求める計算である．要素の比較を並列に行うことができる．kernel3 で用いた方法を使って，1 ピクセル毎に配列要素をまとめて比較し，最小値を画面に出力する．この時，最小要素の配列内の位置を記憶しておかなければならないため，色情報として，最小値と配列の位置にあたるピクセルのテクスチャ座標を出力する．例えば各ピクセルのテクスチャ座標が，左隣のピクセルより 8 大きくなるように設定した場合，1 ピクセル毎に 8 個の配列要素のうち最小値を出力する．以降テクスチャを貼り付けるポリゴンの幅を 8 分の 1 にして，同じ操作を繰り返す．テクスチャが幅 1 高さ 1 になったとき，最小値と最小値の配列内の位置がテクスチャに格納されている．

4.3 2 次元ポアソン方程式

一定の 2 次元領域での温度や電位などの分布が，境界での条件が与えられたとき，平衡状態で満足する方程式は，ポアソン方程式

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho(x, y) \tag{1}$$

で表される．式 (1) は，温度についていえば，$-\rho(x, y)$ の熱源分布がある場合に相当する．正方領域 $[0, 1] \times [0, 1]$ での温度分布を考える．境界条件は，境界における u の値そのものが与えられるディリクレ境界条件とし，領域の上辺の温度が 100 ℃，それ以外の温度が 0 ℃ に固定されているというもの，つまり

$$\begin{aligned} u(0, y) = 0, \quad u(1, y) = 0 \quad (0 < y < 1) \\ u(x, y) = 0, \quad u(x, 1) = 0 \quad (0 < x < 1) \end{aligned} \tag{2}$$

を考える．

4.3.1 反復解法

x, y 方向をそれぞれ h 刻みで $N = 1/h$ 等分し，

$$x_i = ih$$

$$y_i = jh$$

$$\rho_{ij} = \rho(x_i, y_i)$$

として，$u(x_i, y_j)$ に対する近似値を u_{ij} とおき，偏導関数を

$$\frac{\partial^2 u}{\partial x^2} \to \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} \tag{3}$$

$$\frac{\partial^2 u}{\partial y^2} \to \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} \tag{4}$$

と差分近似すれば，

$$\begin{aligned} \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} = \rho(x,y)u_{i,j} &= \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4h^2\rho_{ij}) \\ & (i,j = 1,2,\cdots,N-1) \end{aligned} \tag{5}$$

という $(N-1)^2$ 次の連立方程式が得られる．これは，u_{ij} が周囲の u の平均値から $h^2\rho_{ij}/4$ を引いて得られることを示す．したがって，$u_{ij}^{(0)}$ から出発して，それを式 (4) の右辺に代入して得られる u_{ij} で置き換える，ということを反復すれば，u_{ij} の値はいつか一定の値に収束する．

4.3.2 GPU による実装

反復法としてヤコビ法を用いる．$N \times N$ のサイズを持ち，境界部分のテクセル値以外は全てテクセル値を 0 としたテクスチャを用意する．各テクセルが，u_{ij} に対応する．テクスチャを $N \times N$ のサイズの矩形ポリゴンに貼り付ける．境界条件より境界の値は変化しないため，描画領域を境界を除いた部分にクリップすることで，境界部分の計算を行わないようにしておく．頂点プログラムで，u_{ij} の周囲にある 4 個のテクセルのテクスチャ座標を計算する．次にフラグメントプログラムで，フラグメント毎に対応するテクセルの値を初期値として，式 (4) を計算する．各フラグメントについて，計算結果と，テクセルの値の差をとり，式 (4) の計算結果とこの近似値の残差をピクセルの色としてピクセルバッファに出力する．ピクセルバッファのクリップした部分だけをテクスチャにコピーする．つまり，境界部分のテクスチャは更新されない．次に，L.L.N.L.fortran kernels の 3 番と同様の方法で，各テクセルに格納されている u_{ij} の残差を全て足し合わせ，CPU のメモリに読み出す．残差が 0 に十分近い値であるかを CPU で判定する．残差が 0 に十分近ければ，近似値は収束していると考えられる．近

似値が収束していなければ，ポリゴンを再描画して同じ計算を反復する．収束したと判定されたら，フレームバッファに対してテクスチャを貼り付けた矩形ポリゴンを描画して計算結果を表示するか，CPU がピクセルバッファの内容を読み込み，計算を終了する．収束判定を CPU で行うのは，GPU 側から CPU が行う描画コマンドの発行を中止させることができないためである．

4.3.3 3 次元ポアソン方程式への拡張

3 次元ポアソン方程式は，

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = \rho(x, y, z) \tag{6}$$

で表される．領域の分割幅を N とすると，2 次元の場合と同様に

$$\begin{aligned} u_{i,j,k} &= \frac{1}{6}(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - 6h^2\rho_{ij}) \\ & (i, j, k = 1, 2, \cdots, N-1) \end{aligned} \tag{7}$$

という $(N-1)^3$ 次の連立方程式が得られる．これは，u_{ijk} の前後左右上下の 6 個の値の平均値から $\frac{h^2\rho_{ijk}}{6}$ を引いたものを，新しい近似値として用いることになる．

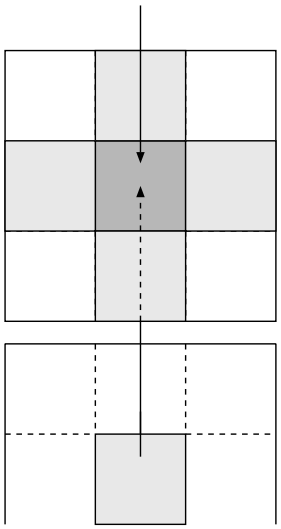

図 6: 3 次元ポアソン方程式

初期値を設定した $N \times N \times N$ のサイズの 3 次元テクスチャを用意する．矩形ポリゴンに，テクスチャ座標が z=0 である 3 次元ポリゴンの断面を貼り付け，式 (5) を計算する．次に 2 次元の場合と同様に，式 (5) の計算結果と近似値の残

差をピクセルバッファに出力し，ピクセルバッファの内容を3次元テクスチャのz=0の平面にコピーする．全てのテクセルの残差を足し合わせた結果を，CPUのメモリに送る．この計算をz=Nの平面まで逐次的に行う．テクスチャの断面毎に計算したテクセルの残差の合計を，CPUで足し合わせ，収束判定を行う．これを1回の反復として，残差が収束するまで計算を行う．2次元における実装では，ヤコビ法を用いて全てのテクセルの値を初期値を元に一斉に更新していたのに対して，3次元における実装では，全ての u_{ijk} の値を並列に更新できない．これは，ピクセルバッファは本質的に2次元であり，3次元テクスチャのテクセル値を，ピクセルバッファの内容を用いて一斉に更新することは不可能であるためである．1枚の断面内ではヤコビ法を用いてテクセルの値が更新し，次の隣接する断面では更新された値を用いて近似値の計算が行われる．どちらの方法で残差は0に収束していく．GeForce FX 5900 Ultraでは，32ビット浮動小数点フォーマットの環境では3次元テクスチャを使用できない. 現時点で実装する場合は固定小数点フォーマットを用いることになるため，精度の高い解を得ることができないと考えられる．

第5章　考察

5.1　L.L.N.L. fortran kernels

L.L.N.L fortran kernelsの24個のカーネルは，以下のように分類できる．

1. ループのステップ間に依存がなく，並列に処理でき．一回のレンダリングで計算が終了する．
2. ループを並列に処理できるが，ループの各ステップ内に逐次計算があるか，出力する値がRGBアルファの4値より多く一度に出力できないため，1回のレンダリングでは計算が終了しない．
3. ループのステップ間に依存があり，1ステップ毎にレンダリングを行う必要がある．
4. プログラムの実行中に，条件に従って計算が停止し，計算結果を格納する配列要素の個数と位置がプログラム内で動的に決定される．

24個のカーネルを上述のパターンに振り分けると，表3のようになる．

パターン	該当カーネル番号
1	1,2,7,8,9,12,15,16,21,22,23
2	3,4,6,10,18,24
3	5,11,13,14,19,20,
4	17

表 3: カーネルの分類

実装方法を考案できなかったカーネルは，(4) に分類される 1 個のみであった．このことから，GPU はベクトルプロセッサとして数値計算を行う能力をほぼ備えているといえる．また，L.L.N.L fortran kernels には，ループのステップ間に依存がなく，並列性がある計算が半分以上あり，数値計算には GPU が得意とする計算が多いということが読み取れる．一方，ループ間に依存があり，逐次計算を行わなければならない計算は，GPU で計算する効果が得られないと考えられる．また条件分岐を多用した処理は，条件分岐の中にさらに条件分岐が現れた場合，実行命令数が $O(n^2)$ の割合で増加するため，効率的でない．このような計算は，CPU で計算した方が効率的であると推測される．

5.2 2 次元ポアソン方程式

ヤコビ法を用いた，2 次元ポアソン方程式の差分方程式の解法を，CPU および GPU 双方で実装した．まず，収束するまでの反復回数を，CPU のみを用いて求めた．収束判定条件は，残差が 0.000001 以下とした．CPU および GPU 双方で，近似解を求める計算を，CPU で求めた回数分繰り返し，実行時間を計測した．図 7 は GPU を用いた計算結果を可視化したものである．データ精度は双方 32 ビット IEEE 浮動小数点を用いた．なお，実行時間の計測の際，CPU は収束判定を行わないようにした．表 4 に実行時間を示す．

CPU を用いたヤコビ法の実装では，同じ内容の配列を 2 つ用意し，1 回の反復毎に更新する配列を入れ替えている．GPU を用いた実装では，ピクセルの R 値の計算のみを近似値の計算に使用し，RGB アルファのベクトル計算は使用していない．32×32 の場合にわずかに CPU の実行時間が GPU より短くっており，

領域分割数 N	CPU 実行時間 (sec)	GPU 実行時間 (sec)	反復回数
32×32	0.0377	0.383	2435
64×64	0.591	0.523	9276
128×128	8.970	8.383	33913
256×256	130.285	165.342	121364

表 4: 反復計算の実行時間

256×256 の場合に CPU の実行時間が GPU の $\frac{4}{5}$ となっている．これは配列のサイズが大きい場合は，GPU のキャッシュミスヒットが生じている為と推測される．また GPU には，ポリゴンを描写し，テクスチャをコピーするオーバーヘッドが存在する．計測結果とこれらの条件から，近似解を求める反復計算において，適切な分割数を選択すれば，GPU は CPU より高速であるといえる．1 反復毎に収束判定を行った場合を考える．GPU から CPU への残差の送信時間は，残差が高々1 ピクセルのデータ量であるため，ボトルネックにはならないと予想されるが，GPU で各テクセルの残差を合計する計算は，複数回レンダリングを繰り返す必要があるため，ボトルネックになると予想される．そのため，1 回毎に収束判定を行うのではなく，数回毎に収束判定を行ったほうが，収束後にさらに反復することがあったとしても，実行速度が向上すると予想される．GPU を用いたシミュレーションにおいて，結果の精度よりも，結果の提示のリアルタイム性が優先される場合は，厳密に収束するのを待つのではなく，収束判定を行わず，一定回数以上反復したら収束したと見なし，計算を打ち切ってもよいと考えられる．Pentium4 プロセッサは，SSE (Streaming SIMD Extensions) と呼ばれる，複数の浮動小数点数演算を同時に実行する機能を持つ．GPU が RGB アルファのベクトル演算を用いた場合と，CPU が SSE に相当する機能を用いた場合の比較は今後の課題である．

5.3 GPU に関する今後の展望

GPU の今後の展望について考察する．

32 ビット浮動小数点フォーマットの制約の撤廃 本研究で使用した GeForce FX

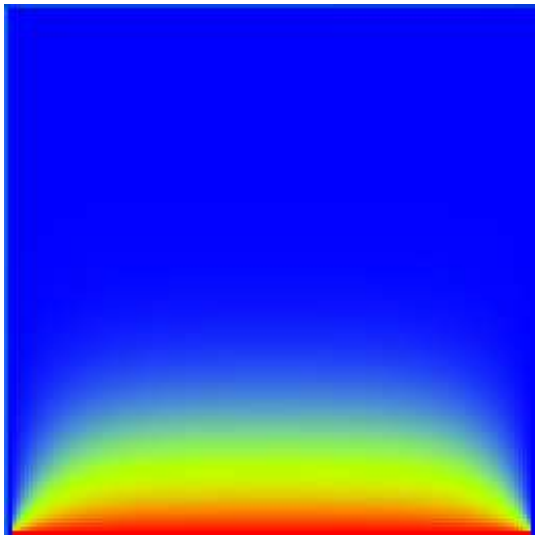

図 7: 2 次元ポアソン方程式の解

5900 Ultra は，トランジスタ数の制約から，32 ビット IEEE 浮動小数点の精度で計算を行う場合は，フラグメントプロセッサのパイプライン数が 8 から 4 に減少する．そのほか，2 次元テクスチャしか使用できない，ポリゴンの色をアルファ値を基に混合するアルファブレンディングなどの機能が使用できないといった様々な制約がある．これらの制約は，トランジスタ数が，32 ビット浮動小数点フォーマットで計算するには，不十分であることが原因である．将来トランジスタ数が増加すれば，これらの制限は撤廃されると予想される．

シェーダ言語の機能拡張　GPU は，DirecX の仕様に追従する形で，機能を向上させている．DirectX の将来のバージョンである DirectX10 では，頂点プログラムとフラグメントプログラムの差異が小さくなり，頂点プログラムからテクスチャにアクセスできるようになる予定である．1 回のレンダリングで実行できるプログラムの長さも増加しつづけており，シェーダ言語はより汎用的な計算を行えるようになると予想される．

CPU-GPU 間の転送速度の改善　PCI-EXPRESS という新しいバス規格が生まれている．PCI-EXPRESS×16 は，AGP×8 と比較すると，データの上り/下りが共に 2 倍の速度を持つ．CPU-GPU 間のデータ転送速度が向上し，頻繁にデータを交換することが可能になると推測される．特に，GPU から CPU へのデータ転送速度が向上することで，GPU の計算結果を CPU が受

け取り，計算を続けることが容易になると推測される，

将来的には，GPU は CG 処理を行う以外に，CPU より高速にベクトル演算を行うベクトルプロセッサとして，CPU と連携して動作するようになると推測される．

第 6 章　おわりに

本研究では,GPU を用いた L.L.N.L. fortran kernels の実装の検討を通して，一般的な数値計算を GPU を用いて行う手法について考察した．さらに，2 次元ポアソン方程式の反復解法を，GPU を用いて実装し,GPU に適した計算を行う場合，GPU は CPU より高速であることを確認した．CPU と GPU が連携して数値計算を行う場合の連携方法が今後の課題である．

謝辞

本研究の機会を与えていただいた富田眞治教授に深甚なる謝意を表します．また，本研究に関して適切なご指導を賜った森眞一郎助教授，中島康彦助教授，五島正裕助手，津邑公暁助手に心から感謝いたします．

さらに，日頃からご討論頂く京都大学情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝いたします．

参考文献

[1] Bolz, J., Farmer, I., Grinspun, E. and Schoder, P. : Sparse Matrix Solvers on the GPU:Conjugate Gradients and Multigrid, *SIGGRAPH 03 Proceedings* (2003).

[2] Thompson, C., Hahn, S. and Oskin, M. : Using Modern Graphics Architectures for General-Purpose Computing : A Framework and Analysis, *Proceedings of 35th International Symposium on Microarchitecture (MICRO-35)* , pp. 306–320 (2002).

[3] Krüger, J. and Westermann, R. : Linear Algebra Operators for GPU Implementation of Numerical Algorithms, *SIGGRAPH 03 Proceedings* (2003).

[4] Larsen, E. and McAllister, K. : Fast Matrix Multiplies using Graphics Hardware, *In Proceedings Supercomputing 2001* (2001).

[5] Ertl, T., Weiskopf,D., Kraus, M., Engel,K., Weiler, M., Hopf, M., Rottger, S., and Rezk-Salama C. : "Programmable Graphics Hardware for Interactive Visualization," Eurographics2002 チュートリアルノート (T4) (2002).

[6] NVIDIA Corpoaration : Cg Toolkit User's Manual Release 1.1 (2003).

[7] NVIDIA Corpoaration : NVIDIA OpenGL Extension Specifications (2003).

[8] NVIDIA Corpoaration : NVIDIA GEFORCE FX 5900 PRODUCT OVERVIEW (2003).

[9] Woo, M. and Neider, J. and Davis, T. (株式会社アクロス訳) : OpenGL プログラミングガイド第 2 版, ピアソンエデュケーション (1997).

[10] J. A. エドワード (滝沢, 牧野訳) : OpenGL 入門, ピアソンエデュケーション (2002).

[11] Z, Zima. and Chapman, B. (村上訳) : スーパーコンパイラ, オーム社, pp. 195–272 (1995).

[12] 伊藤正夫, 藤野和建 : 数値計算の常識, 共立出版, pp. 130–133 (1985).

[13] 富田眞治 : 並列コンピュータ工学, 昭光堂 (1996).