삼중대각행렬 시스템 풀이의 빠른 GPU 구현
(Fast GPU Implementation for the Solution of Tridiagonal Matrix Systems)

김 영 희 † 이 성 기 ††
(Yonghee Kim) (Sungkee Lee)

요 약 컴퓨터 하드웨어의 급속한 발전으로 그래픽 프로세서 유닛(Graphics Processor Units : GPUs)은 굉장한 메모리 대역폭과 산술 능역을 보유하게 되어 범용 계산에 많이 활용되고 있으며, 특히 계산 집약적인 물리 기반 시뮬레이션(physics based simulation)의 GPU 구현이 활발하게 연구되고 있다. 물리 기반 시뮬레이션의 기본이 되는 미분방정식 풀이 과정에서 삼중대각행렬(tridiagonal matrix) 시스템은 유한차분(finite-difference) 근사에 의해서 자주 나타나는 선형시스템으로 물리 기반 시뮬레이션 관점에서 삼중대각행렬 시스템의 빠른 풀이는 중요한 연구 분야이다.

본 논문에서는 GPU에서 삼중대각행렬 시스템 풀이를 빠르게 구현할 수 있는 방법을 제안한다. 벡터 프로세서(vector processor) 계산에서 삼중대각행렬 시스템 풀이 방법으로 널리 사용되는 cyclic reduction 또는 odd-even reduction 알고리즘을 GPU에서 구현하였다. 본 논문에서 제안한 방법을 삼중대각행렬 시스템 풀이 방법으로 잘 알려져 있는 Thomas 방법과 GPU를 이용한 선형시스템 풀이에서 좋은 성과를 보이고 있는 conjugate gradient 방법과 비교할 때 상당한 성능 향상을 얻을 수 있었다. 또한, 열전도(heat conduction) 방정식, 이류 확산(advection-diffusion) 방정식, 얕은 물(shallow water) 방정식에 의한 물리 기반 시뮬레이션의 GPU 구현에 본 논문에서 제안한 방법을 사용하여 1024×1024 격자의 계산 영역에서 초당 35프레임 이상의 놀라운 성능을 보여주었다.

키워드 : 그래픽 프로세서 유닛, Cyclic Reduction 방법(Odd-Even Reduction), 삼중대각행렬, 선형시스템 풀이, 물리 기반 시뮬레이션

Abstract With the improvement of computer hardware, GPUs(Graphics Processor Units) have tremendous memory bandwidth and computation power. This leads GPUs to use in general purpose computation. Especially, GPU implementation of compute-intensive physics based simulations is actively studied. In the solution of differential equations which are base of physics simulations, tridiagonal matrix systems occur repeatedly by finite-difference approximation. From the point of view of physics based simulations, fast solution of tridiagonal matrix system is important research field.

We propose a fast GPU implementation for the solution of tridiagonal matrix systems. In this paper, we implement the cyclic reduction(also known as odd-even reduction) algorithm which is a popular choice for vector processors. We obtained a considerable performance improvement for solving tridiagonal matrix systems over Thomas method and conjugate gradient method. Thomas method is well known as a method for solving tridiagonal matrix systems on CPU and conjugate gradient method has shown good results on GPU. We experimented our proposed method by applying it to heat conduction, advection-diffusion, and shallow water simulations. The results of these simulations have shown a remarkable performance of over 35 frame-per-second on the 1024×1024 grid.

Key words : GPU(Graphic Process Units), Cyclic Reduction Method(Odd-Even Reduction), Tridiagonal Matrix, Linear System Solver, Physics Based Simulation

† 정 회 원 : 경북대학교 컴퓨터과학과
yhkim@mail2.knu.ac.kr
†† 종신회원 : 경북대학교 전자전기컴퓨터학부 교수
sklee@knu.ac.kr
논문접수 : 2005년 9월 9일
심사완료 : 2005년 10월 10일

1. 서 론

과거의 그래픽 프로세서(Graphics Processor Units : GPU)는 특수한 목적에 의해 설계되어 3차원 장면을 그리는 등의 그래픽 작업에 대해서는 매우 빠르지만 CPU

가 처리하는 범용적인 계산을 실행할 수 없었다. 그러나 부동소수점 계산과 프로그래밍 가능한 정점 셰이더(vertex shader)와 프래그먼트 셰이더(fragment shader)의 등장으로 GPU는 중요한 계산 자원으로 여겨지기 시작하였다. 그래픽 하드웨어에 대한 프로그램 가능성의 향상과 더불어 그래픽 하드웨어 성능이 급속하게 발전하여 최근의 GPU는 상당한 메모리 대역폭(memory bandwidth)과 산술 연산 능력을 보유하고 있으며 32-bit 부동소수점 계산이 가능하게 되었다. GPU 성능의 향상으로 인하여 GPU를 범용 계산에 활용하려는 연구는 컴퓨터 그래픽스 분야를 넘어 다양한 응용 분야에서 상당한 관심사로 자리 잡고 있으며 GPGPU(General Purpose computation on GPU)로 잘 알려질 만큼 각광을 받고 있다. 그 결과로써 GPU 구현은 랜더링과 가시화[1]처럼 그래픽스 관련 분야 분야뿐만 아니라 미분 방정식의 선형시스템 풀이를 위한 행렬 계산[2-4]에서 물리 기반 시뮬레이션[5-7], 영상 처리[8], 데이타베이스 연산[9]에 이르기 까지 다양하다. 많은 개발자들은 그래픽 하드웨어가 CPU에서 수행되던 문제들을 풀 수 있기를 기대하고 있으며 다양한 분야에서 상당한 성과를 보여주고 있다.

GPU에서 범용 계산은 데이타를 텍스쳐로 표현하는데, 이런 텍스쳐는 배열(array)로써 여겨질 수 있으므로 행렬 계산이나 데이타가 2차원 또는 3차원 격자(grid)로 표현되는 미분 방정식 계산에 적합하다. 따라서 conjugate gradient와 multigrid 방법 등을 사용하여 희박 선형시스템(sparse linear systems) 풀이에서 성능 향상을 보여주고 있으며[2-4], 특히 물리 기반 시뮬레이션(physics based simulation)에서 GPU 구현은 상당한 성능 향상을 보이고 있다[5-7]. 이러한 이유는 첫째로 물리 기반 시뮬레이션은 미분 방정식의 풀이와 행렬식의 풀이로 대부분 이루어져 있다. 둘째로 GPU 구현은 데이타를 비디오 메모리(video memory)로 전송시키는 등의 계산이외의 작업 부하가 있는데 시뮬레이션에서는 초기에 한번만 처리하면 되므로 이런 부하가 문제시 되지 않는다. 마지막은 시뮬레이션에서는 계산 결과를 시각적으로 가시화하는 경우가 많은데 GPU를 이용한 계산에서는 계산 결과가 이미 비디오 메모리에 저장되어 있기 때문에 가시화 시간을 단축할 수 있다. 이러한 이유들로 인하여 물리 기반 시뮬레이션의 GPU 구현에 대한 활발한 연구가 진행되고 있다.

본 논문에서는 삼중대각행렬(tridiagonal matrix) 시스템 풀이를 GPU에서 빠르게 구현할 수 있는 방법을 제안한다. 삼중대각행렬 시스템은 물리 기반 시뮬레이션의 기본이 되는 미분방정식에서 유한차분 근사(finite-difference approximation)에 의해서 자주 나타나는 중요한 선형시스템이다. 특히 2차원 또는 3차원 미분 방정식의 경우에 오중대각행렬(penta-diagonal matrix) 또는 칠중대각행렬(hept-diagonal matrix)을 풀어야하는 부담을 줄이기 위하여 시간 분할 기법(time-splitting scheme) 또는 ADI(Alternate Direction Implicit) 방법을 자주 사용하는데, 이 때 삼중대각행렬의 선형시스템이 나타난다[10]. 삼중대각행렬 시스템 풀이는 일반적인 행렬 시스템 풀이나 희박행렬 시스템 풀이와는 별개로 취급되어 삼중대각의 특징을 이용한 빠른 풀이 기법들이 제안되어 왔다. 그중에서 시간 복잡도(time complexity)가 O(N)인 Thomas 방법은 CPU에서의 대부분의 계산에 사용되고 있다. 그러나 불행히도 Thomas 방법은 GPU 구현에 적합하지 않아서 GPU 구현으로 인한 성능 향상을 기대하기 어렵다. 그리고 GPU에서의 선형시스템 풀이 방법으로 제안된 conjugate gradient, Jacobi, miltigrid 방법들은 삼중대각행렬의 특성을 잘 반영하지 못하기 때문에 성능이 떨어진다.

본 논문에서는 cyclic reduction 또는 odd-even reduction 알고리즘을 GPU에서 빠르게 구현함으로써 GPU에서 삼중대각행렬 시스템 풀이에 상당한 성능 향상을 얻을 수 있었다. Cyclic reduction 방법은 벡터 프로세서(vector processor) 계산에서 삼중대각행렬 시스템 풀이로 널리 사용되는 방법으로, 이 방법에는 직렬(serial)과 병렬(parallel)로 두 종류의 알고리즘이 있다[11,12]. 본 논문에서는 시간 복잡도가 O(N)으로 빠른 직렬 알고리즘을 사용하였다. 본 논문의 방법을 CPU에서의 Thomas 방법과 GPU를 이용한 행렬 시스템 풀이로 잘 알려진 conjugate gradient 방법과 비교할 때 계산 성능이 상당히 향상되었다. 또한, 열전도(heat conduction) 방정식, 이류 확산(advection-diffusion) 방정식, 얕은 물(shallow water) 방정식에 의한 물리 기반 시뮬레이션의 GPU 구현에 본 논문에서 제안한 방법을 사용하여 1024×1024 격자의 계산 영역에서 초당 35프레임 이상의 놀라운 성능을 보여주었다.

논문의 구성은 다음과 같다. 2장에서는 미분 방정식에서 삼중대각행렬 시스템이 나타나는 예를 보여준다. 3장에서는 cyclic reduction 알고리즘을 소개하고 cyclic reduction 알고리즘의 GPU 구현을 설명한다. 4장에서는 삼중대각행렬 시스템의 풀이 방법들에 대한 성능을 비교하고 삼중대각행렬 시스템 풀이를 이용한 물리 기반 시뮬레이션의 구현 결과를 보여준다. 마지막 5장에서는 결론 및 향후 연구에 대해서 설명한다.

2. 삼중대각행렬

모든 물리현상을 지배하는 지배방정식은 미분 방정식의 형태로 나타낼 수 있다. 삼중대각행렬 방정식은 이차

미분을 가진 미분 방정식에 대한 유한 차분 근사에 의해서 자주 나타나는 중요한 방정식이다. 단순 조화 운동(simple harmonic motion), Helmholtz, Laplace, Poisson, 확산 방정식 등이 그 예이다. 따라서 이런 방정식의 풀이에 대한 효율적인 방법은 매우 중요한 알고리즘이다. 이차 미분을 가진 미분 방정식을 암시적 방법(implicit scheme)을 이용하여 유한 차분하는 경우 오중대각행렬 또는 칠중대각행렬의 선형 시스템이 생기는데 이를 풀기 위해서는 상당한 계산을 필요로 한다. 그러나 시간 분할 기법 또는 ADI 방법을 이용하여 2차원의 경우 x-방향과 y-방향에 대해서 각각의 삼중대각행렬 시스템으로 만들면 효율적으로 풀 수 있다. 다음은 미분 방정식의 유형별로 삼중대각행렬 시스템이 나타나는 기본적인 예를 설명할 것이다.

2.1 열전도 방정식(heat conduction equation) : 포물선형 방정식

경계층 유동 방정식, 포물선화(parabolized) Navier-Stokes 방정식 등이 포함되는 포물선형 편미분 방정식의 모델 방정식은 비정상 상태 열전도 방정식(unsteady state heat conduction equation)이다[13]. 비정상 상태의 열전도 방정식은 고체와 같이 흐름이 없는 곳에서의 열의 이동을 나타내는 방정식으로 2차원 방정식의 형태는 다음과 같다.

$$\frac{\partial T}{\partial t}=c\left(\frac{\partial T^2}{\partial x^2}+\frac{\partial T^2}{\partial y^2}\right) \quad (1)$$

여기서, T는 온도를 의미하며 c는 열 확산 계수 이다. 식 (1)에 완전 암시적 방법(fully implicit scheme)을 적용시키면 다음의 편미분 방정식을 얻을 수 있다.

$$\frac{T_{i,j}^{n+1}-T_{i,j}^{n}}{\Delta t}=c\left(\frac{T_{i+1,j}^{n+1}-2T_{i,j}^{n+1}+T_{i-1,j}^{n+1}}{\Delta h^2}+\frac{T_{i,j+1}^{n+1}-2T_{i,j}^{n+1}+T_{i,j-1}^{n+1}}{\Delta h^2}\right)$$

방정식을 정리하여 행렬방정식을 만들면 좌변에 5개의 항이 나타나고 다음과 같은 오중대각행렬 방정식이 만들어 진다.

$$AT_{i-1,j}^{n+1}+BT_{i,j}^{n+1}+CT_{i+1,j}^{n+1}+DT_{i,j-1}^{n+1}+ET_{i,j+1}^{n+1}=-T_{i,j}^{n} \quad (2)$$

여기서 A, B, C, D, E는 Δt, Δh, c에 관한 상수이다. 그러나 식 (1)에 ADI 방법을 이용하여 시간 미분의 수치 적분을 하면 다음 식 (3)과 같이 2차원의 경우 x-방향과 y-방향에 대해서 각각 삼중대각행렬 방정식을 얻을 수 있다. ADI 방법은 2차원의 경우 두 단계(3차원의 경우 세 단계)로 이루어진다[10].

• 단계 1 (x-방향) (3)

$$\frac{T_{i,j}^{n+1/2}-T_{i,j}^{n}}{\Delta t/2}=c\left(\frac{T_{i+1,j}^{n+1/2}-2T_{i,j}^{n+1/2}+T_{i-1,j}^{n+1/2}}{\Delta h^2}+\frac{T_{i,j+1}^{n}-2T_{i,j}^{n}+T_{i,j-1}^{n}}{\Delta h^2}\right)$$

• 단계 2 (y-방향)

$$\frac{T_{i,j}^{n+1}-T_{i,j}^{n+1/2}}{\Delta t/2}=c\left(\frac{T_{i+1,j}^{n+1/2}-2T_{i,j}^{n+1/2}+T_{i-1,j}^{n+1/2}}{\Delta h^2}+\frac{T_{i,j+1}^{n+1}-2T_{i,j}^{n+1}+T_{i,j-1}^{n+1}}{\Delta h^2}\right)$$

식 (3)의 단계 1에서는 이미 알고 있는 $T_{i,j}^{n}$의 데이타를 이용하여 $T_{i,j}^{n+1/2}$를 구하는 것으로 각각의 j -격자선에 대해서 단계 1의 방정식을 적용시키면 $T_{i,j}^{n+1/2}$에 대해서 식 (4), 식 (5)와 같은 삼중대각행렬 방정식을 얻게 되고 이로 부터 $T_{i,j}^{n+1/2}$를 구할 수 있다.

$$A_iT_{i-1,j}^{n+1/2}+B_iT_{i,j}^{n+1/2}+C_iT_{i+1,j}^{n+1/2}=K_i \quad (4)$$

$j=1,..,N$

$$\begin{bmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & & \ddots & & \\ & & A_{N-1} & B_{N-1} & C_{N-1} \\ & & & A_N & B_N \end{bmatrix} \bullet \begin{bmatrix} T_{1,j}^{n+1/2} \\ T_{2,j}^{n+1/2} \\ \vdots \\ T_{N-1,j}^{n+1/2} \\ T_{N,j}^{n+1/2} \end{bmatrix} = \begin{bmatrix} K_1 \\ K_2 \\ \vdots \\ K_{N-1} \\ K_N \end{bmatrix} \quad (5)$$

식 (3)의 단계 2에서는 단계 1에서 구한 $T_{i,j}^{n+1/2}$를 이용하여 $T_{i,j}^{n+1}$를 구하는 것으로 각각의 i-격자 선에 대해서 단계 2의 방정식을 적용시키면 역시 $T_{i,j}^{n+1}$에 대해서 삼중대각행렬 방정식을 얻게 되고 $T_{i,j}^{n+1}$을 구할 수 있다.

2.2 이류 확산 방정식(advetion-diffusion equation) : 쌍곡선-포물선형 방정식

이류 확산 방정식은 쌍곡선-포물선(hyperbolic and parabolic)형 미분 방정식으로 대기 경계층(planetary boundary layer)에서 오염.물질의 평균 농도를 예측하기 위한 기능성 대기 분산 모델에 널리 적용된다. 2차원 이류 확산 방정식은 다음과 같다.

$$\frac{\partial C}{\partial t}+u\frac{\partial C}{\partial x}+v\frac{\partial C}{\partial y}=D\left(\frac{\partial C^2}{\partial x^2}+\frac{\partial C^2}{\partial y^2}\right)+S(x,y,t) \quad (6)$$

식 (6)에서 C는 오염 물질 농도(pollutant concentration)를 D는 확산 계수를 나타내며 (u,v)는 x-축과 y-축 방향의 속도를 $S(x,y,t)$는 외부에서 가해지는 오염 물질의 농도를 의미한다. 여기에 암시적 방법을 적용시키면 다음의 편미분 방정식을 얻을 수 있다.

$$aC_{i-1,j}^{n+1}+bC_{i+1,j}^{n+1}+dC_{i,j-1}^{n+1}+eC_{i,j+1}^{n+1}+fC_{i,j}^{n+1}=C_{i,j}^{n}+S_{i,j}\Delta t \quad (7)$$

$a=-u_{i,j}\frac{\Delta t}{2\Delta h}-\alpha$, $b=u_{i,j}\frac{\Delta t}{2\Delta h}-\alpha$,

$$d=-v_{i,j}\frac{\Delta t}{2\Delta h}-\alpha,\quad e=v_{i,j}\frac{\Delta t}{2\Delta h}-\alpha,$$

$$f=1+4\alpha,\quad \alpha=D\Delta t/\Delta h^2$$

열전도 시뮬레이션에서 설명하였듯이 오중대각행렬의 풀이를 필요로 한다. 또한, 식 (7)의 오중대각행렬의 선형시스템은 조건부 안정(conditionally stable)하다. 즉, $u_{i,j}$, $v_{i,j}$, Δt, Δh, D 등의 상수 값에 따라 시스템이 진동하거나 발산하여 해를 찾을 수 없는 경우가 발생함을 의미한다. 이에 대해서 식 (6)에 ADI 방법을 사용하고 풍상(upwind) 미분 연산자를 사용하여 다음 식 (8)과 같이 시스템을 무조건 안정(unconditionally stable)하게 만들면서 삼중대각행렬 시스템으로 만들 수 있다[14].

$$\frac{C_{i,j}^{n+1/2}-C_{i,j}^{n}}{\Delta t/2}=-u_{i,j}\Delta_x^{*}C_{i,j}^{n+1/2}-v_{i,j}\Delta_x^{0}C_{i,j}^{n}+D\left(\Delta_{xx}C_{i,j}^{n+1/2}+\Delta_{yy}C_{i,j}^{n}\right)+S_{i,j}$$

$$\frac{C_{i,j}^{n+1}-C_{i,j}^{n+1/2}}{\Delta t/2}=-u_{i,j}\Delta_x^{0}C_{i,j}^{n+1/2}-v_{i,j}\Delta_x^{*}C_{i,j}^{n+1}+D\left(\Delta_{xx}C_{i,j}^{n+1/2}+\Delta_{yy}C_{i,j}^{n+1}\right)+S_{i,j}\quad(8)$$

Δ_{yy}와 Δ_{xx}는 2차 미분 연산자이고 Δ^0는 중앙 미분 연산자이며 Δ^*는 풍상미분 연산자이다. 식 (8)의 행렬 시스템은 무조건 안정하기 때문에 시뮬레이션에서 큰 시간 간격(time step, Δt)에 대해서 계산이 가능하여 실시간 처리가 필요한 경우에 유용한 모델이다.

2.3 Laplace 방정식과 Poisson 방정식 : 타원형 방정식

타원형 방정식은 Laplace 방정식과 Poisson 방정식으로 대표되는 편미분 방정식으로 공학 분야에서 광범위하게 나타나는 방정식으로 다음과 같다.

$$\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}=g(x,y)\quad(9)$$

선 Gauss-Seidel 반복법, 선 SOR(Line Successive Over-Relaxation), ADI와 같은 선 반복법(line-iterative methods)을 사용하여 풀이할 때 삼중대각행렬 방정식이 나타난다. 예로 ADI 방법을 직접 사용하는 경우는 다음과 같은 두 단계를 수렴할 때가지 반복하여 계산하는 것으로 각 단계는 삼중대각행렬 방정식이다.

• 단계 1

$$\frac{f_{i+1,j}^{n+1/2}-2f_{i,j}^{n+1/2}+f_{i-1,j}^{n+1/2}}{\Delta h^2}+\frac{f_{i,j+1}^{n}-2f_{i,j}^{n}+f_{i,j-1}^{n}}{\Delta h^2}=g(x,y)$$

• 단계 2

$$\frac{f_{i+1,j}^{n+1/2}-2f_{i,j}^{n+1/2}+f_{i-1,j}^{n+1/2}}{\Delta h^2}+\frac{f_{i,j+1}^{n+1}-2f_{i,j}^{n+1}+f_{i,j-1}^{n+1}}{\Delta h^2}=g(x,y)$$

3. Cyclic reduction

3.1 Cyclic reduction 알고리즘

Cyclic reduction 방법은 삼중대각행렬 방정식을 풀기 위해서 Hockney에 의해서 제안되었으며[11], 벡터 프로세서(vector processor) 계산에서 삼중대각행렬 시스템 풀이로 널리 사용되는 방법이다. 알고리즘은 두 종류가 있는데 하나는 odd-even reduction 또는 직렬 cyclic reduction이고 다른 하나는 odd-even elimination 또는 병렬 cyclic reduction이다[12]. 본 논문에서는 두 알고리즘 중에서 산술연산이 적은 직렬 cyclic reduction 알고리즘을 GPU로 구현하였다. 직렬 cyclic reduction 알고리즘은 순차(sequential) 알고리즘에 병렬성을 증대시키기 위하여 행렬시스템을 재정렬 한 것으로 O(N) 시간 복잡도를 가진다. 병렬 cyclic reduction은 O(NlogN) 산술연산을 수행하는 알고리즘으로 N개의 프로세서에서 O(logN) 산술 연산을 수행할 때 선호되는 알고리즘으로 본 논문에서는 자세한 설명은 생략한다.

다음은 1차원 미분 방정식에서 유도된 삼중대각행렬의 방정식을 cyclic reduction 방법으로 해를 구하는 과정을 설명한다. Cyclic reduction 방법은 방정식의 수가 2의 배수이어야 하는 제약 조건이 있다. 이런 제약 조건이 필요 없는 방법들이 제안되었지만, 본 논문에서는 방정식의 수 N을 다음과 같이 가정한다.

$$N=N'-1,\ N'=2^q\ (q\text{는 정수})$$

실제 방정식이 아닌 끝 부분의 방정식을 다음과 같이 설정하면

$$x_0=x_{N'}=0$$

$i=2,4,\ldots,N'-2$에 대해서 세 개의 인접한 방정식은 다음과 같이 나타낼 수 있다.

$$\begin{aligned}a_{i-1}x_{i-2}+b_{i-1}x_{i-1}+c_{i-1}x_i&=k_{i-1}\\a_ix_{i-1}+b_ix_i+c_ix_{i+1}&=k_i\\a_{i+1}x_i+b_{i+1}x_{i+1}+c_{i+1}x_{i+2}&=k_{i+1}\end{aligned}\quad(10)$$

첫 번째 방정식에 $\alpha_i=-a_i/b_{i-1}$를 곱하고 세 번째 방정식에 $\beta_i=-c_i/b_{i+1}$를 곱하여 세 개의 방정식을 더하면 x_{i-1}와 x_{i+1}항이 제거되고 다음과 같은 방정식을 얻는다.

$$a_i^{(1)}x_{i-2}+b_i^{(1)}x_i+c_i^{(1)}x_{i+2}=k_i^{(1)},\ i=2,4,\ldots,N'-2\quad(11)$$

$$a_i^{(1)} = \alpha_i a_{i-1}$$
$$c_i^{(1)} = \gamma_i c_{i+1}$$
$$b_i^{(1)} = b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1}$$
$$k_i^{(1)} = k_i + \alpha_i k_{i-1} + \gamma_i k_{i+1}, \quad \alpha_i = -a_i / b_{i-1}, \quad \beta_i = -c_i / b_{i+1}$$

식 (11)은 모두 짝수 변수로만 이루어져 있고, 계수 $(a_i^{(1)}, b_i^{(1)}, c_i^{(1)}, k_i^{(1)})$를 가지는 것을 제외하고는 원 방정식과 같은 삼중대각행렬의 방정식이다. 그러나 방정식의 수는 반으로 줄었다. 이 과정을 축소(reduction) 과정이라 하자. 이 축소 과정을 다음 식 (12)와 같이 $i = N'/2$에 대한 방정식만 남을 때까지 $\log_2^{N'} - 1$번 반복하고 계수들을 저장한다.

$$a_{N'/2}^{(s)} x_0 + b_{N'/2}^{(s)} x_{N'/2} + c_{N'/2}^{(s)} x_{N'} = k_{N'/2}^{(s)} \tag{12}$$

여기서 s는 $\log_2^{N'} - 1$이고 축소 과정의 레벨(level)을 의미한다. 여기서 $x_0 = x_{N'} = 0$이기 때문에 $i = N'/2$에 대한 방정식의 해는 다음과 같이 얻어진다.

$$x_{N'/2} = k_{N'/2}^{(s)} \Big/ b_{N'/2}^{(s)} \tag{13}$$

이 과정을 채우기(filling) 과정이라 하자. 다음 $s-1$ 레벨에서는 x_0, $x_{N'}$, $x_{N'/2}$을 알고 있으므로 다음 식 (14)를 사용하여 해를 구할 수 있다.

$$x_i = \left(k_i^{(s-1)} - a_i^{(s-1)} x_{i-N'/4} - c_i^{(s-1)} x_{i+N'/4}\right) / b_i^{(s-1)},$$
$$i = N'/4, \quad 3N'/4 \tag{14}$$

남은 방정식의 해도 이 채우기 과정을 반복함으로써 구할 수 있다. 전체 알고리즘의 의사코드는 다음 식 (15), (16)과 같다. 그림 1은 벡터 $P_i^{(s)} = (a_i^{(s)}, b_i^{(s)}, c_i^{(s)}, k_i^{(s)})$를 정의할 때 $N = 15 = 2^4 - 1\ (q = 4)$에 대해서 cyclic reduction 방법으로 해를 구하는 과정을 다이어그램으로 보여준다.

• 축소 과정

$for\ s \leftarrow 1\ to\ q-1$

$for\ i \leftarrow 2^s\ to\ N' - 2^s,\ step = 2^s$

$$a_i^{(s)} = \alpha_i a_{i-2^{(s-1)}}^{(s-1)}$$
$$c_i^{(s)} = \gamma_i c_{i+2^{(s-1)}}^{(s-1)}$$
$$b_i^{(s)} = b_i^{(s-1)} + \alpha c_{i-2^{(s-1)}}^{(s-1)} + \gamma a_{i+2^{(s-1)}}^{(s-1)}$$
$$k_i^{(s)} = k_i^{(s-1)} + \alpha k_{i-2^{(s-1)}}^{(s-1)} + \gamma k_{i+2^{(s-1)}}^{(s-1)}$$

$where$ (15)

$$\alpha_i = -a_i^{(s-1)} \Big/ b_{i-2^{(s-1)}}^{(s-1)}, \quad \gamma_i = -c_i^{(s-1)} \Big/ b_{i+2^{(s-1)}}^{(s-1)}$$

• 채우기 과정

$for\ s \leftarrow q\ to\ 1,\ \ step = -1$

$for\ i \leftarrow 2^{s-1}\ to\ N' - 2^{s-1},\ step = 2^{s-1}$

$$x_i = \left(k_i^{(s-1)} - a_i^{(s-1)} x_{i-2^{(s-1)}} - c_i^{(s-1)} x_{i+2^{(s-1)}}\right) / b_i^{(s-1)} \tag{16}$$

3.2 Cyclic reduction의 GPU 구현

본 논문에서는 OpenGL API와 Cg 셰이딩 언어(shad-

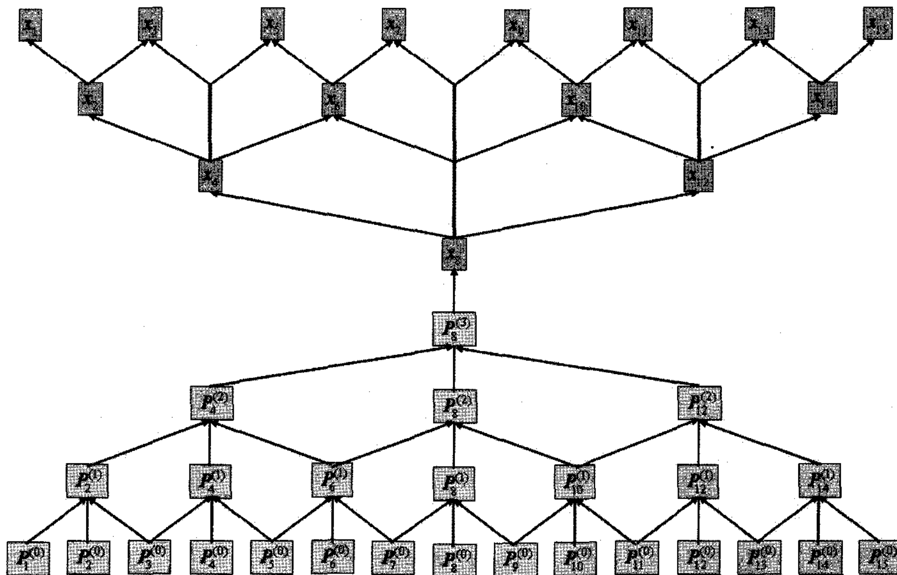


그림 1 N=15인 경우에 cyclic reduction 방법으로 해를 구하는 과정

ing language)로 프래그먼트 셰이더(fragment shader)를 사용하여 구현하였다.

Cyclic reduction 방법의 레벨 단위별 계산을 수행하기 위해서 OpenGL의 pbuffer(pixel buffer)와 ping pong 기법을 이용하였다. pbuffer는 오프스크린 프레임 버퍼(offscreen frame buffer)로써 여러 면(multi-surface)을 가질 수 있다. pbuffer의 면(surface)은 front, back, aux0, aux1등의 컬러 버퍼(color buffer)를 의미한다. 다시 말하면 pbuffer는 여러 컬러 버퍼를 가질 수 있다. 각 면은 각각의 랜더링 가능한 텍스쳐를 가지며 이들의 전환(switching)은 매우 빠르다. 또한 pbuffer는 오프스크린 랜더링 타겟(offscreen rendering target)의 사용을 허락한다. 랜더링 타겟은 중간 단계의 이미지를 저장하기 위한 컬러 버퍼로써 사용이 가능하고, 또한 입력 텍스쳐로도 사용이 가능한 비디오 메모리 영역이다. 랜더링 타겟이 다음 단계에 입력 텍스쳐로 사용이 가능하여 이런 과정을 RTT(Render-To-Texture)라 한다. ping pong 기법은 pbuffer의 이런 특징을 이용한 것으로, 랜더링 결과를 하나의 컬러 버퍼에 쓰고 다음 단계에 이 컬러 버퍼를 입력 텍스쳐로 사용하여 데이타를 읽어올 수 있다. 이 때 어떤 데이타의 복사나 다른 처리를 필요로 하지 않는다. 반대로 이 입력 텍스쳐였던 컬러 버퍼에 랜더링 결과를 다시 쓸 수 있다. pbuffer와 ping pong 기법을 사용함으로써 다음에서 설명할 cyclic reduction의 빠른 GPU 구현이 가능하다.

다음은 앞에서 예로 든 $N=15=2^4-1$ $(q=4)$인 경우에 대해서 cyclic reduction 방법의 GPU 구현을 다이어그램으로 보여줄 것이다. 그림 2는 계수 P 벡터를 계산하는 축소 과정으로 원 방정식의 계수들은 텍스쳐(texture) P0에 저장하고 각 단계에서 계산된 P 벡터들은 레벨(s) 단위로 하나의 컬러 버퍼에 연속적으로 저장한다. 여기서 P1과 P2는 pbuffer의 면(surface)으로써 컬러 버퍼이며 텍스쳐와 컬러 버퍼의 데이타 형식은 4개의 실수(floating-point) 요소를 가지는 벡터 형식이다. 레벨 1($s=1$)에서는 텍스쳐 P0에서 $P_i^{(0)}$ 데이타를 읽어 와서 식 (15)에 의해 계산하여 P1 버퍼에 쓴다. 레벨 2($s=2$)에서는 P1 버퍼에 저장된 $P_i^{(1)}$ 데이타를 읽어 와서 $P_i^{(2)}$를 계산하여 P2 버퍼에 쓰고, 레벨 3($s=3$)에서는 이 계산된 결과를 다시 읽어 와서 $P_i^{(3)}$를 계산하여 P1 버퍼에 쓴다. 이런 ping pong 과정을 반복하여 계산이 끝나면 데이타를 하나의 버퍼로 모아준다. 그림 2에서는 데이타가 많이 저장되어 있는 버퍼로 모아주었다. $P_i^{(1)}$, $P_i^{(2)}$, $P_i^{(3)}$가 저장되는 위치와 인덱스 계산에 대해서는 뒤에서 설명할 것이다.

그림 3은 원 방정식의 계수인 텍스쳐 P0와 축소 과정에서 계산된 P 벡터가 저장된 텍스쳐 P1으로 부터 해 x를 구하는 채우기 과정이다. 여기서 X1, X2 컬러 버퍼 역시 pbuffer의 면(surface)이고, 데이타 형식은 1개의 실수 요소를 가진다. 계산 과정은 축소 과정과 유사하다. 레벨 4($s=4$)에서는 텍스쳐 P1에서 $P_i^{(3)}$ 데이타를 읽어 와서 식 (16)에 의해 x_8을 계산하여 x의 인덱스 위치에 쓴다. 레벨 3($s=3$)부터는 텍스쳐 P1과 X1 버퍼에서 데이타를 읽어 와서 x_4와 x_{12}를 계산하여 X2 버퍼에 쓴다. 레벨 1($s=1$)까지는 X1 버퍼와 X2 버퍼에서 번갈아 가며 데이타를 읽어오고, 계산된 결과를 X1, X2 버퍼에 번갈아 가며 쓰는 ping pong 과정을 반복한다. 계산이 끝나면 하나의 버퍼로 복사하여 모아준다.

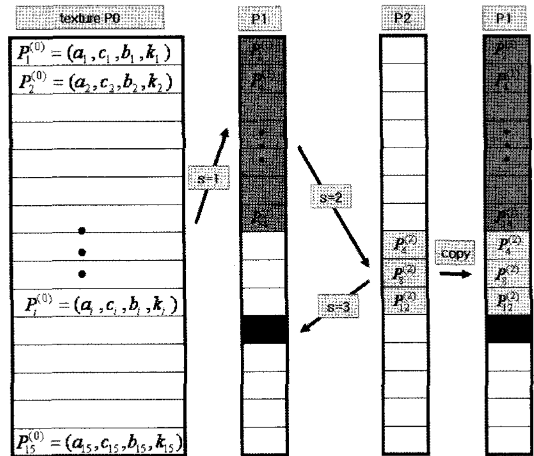


그림 2 GPU에서 계수 P 벡터 계산 과정

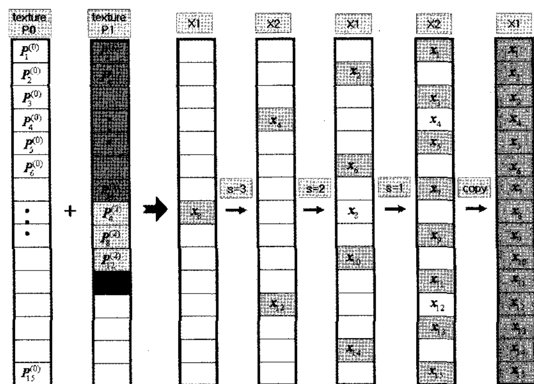


그림 3 GPU에서 해 x 계산 과정

각 레벨에서 계산된 $P_i^{(s)}$ 들을 하나의 버퍼에 저장함으로써 계수 P 벡터와 해 x 계산 과정에서 인덱스 계산이 필요하다. GPU 계산에서는 텍스쳐 좌표 매핑으로 가능하며 그림 4는 계수 P 벡터 계산에서의 좌표 매핑을 보여준다. 본 논문에서는 인덱스 계산을 간단히 하기 위해서 [0, 텍스쳐 너비(texture width)] 범위의 좌표를 사용하는 texRect 형태의 텍스쳐 좌표를 사용한다. [0,1] 범위의 좌표를 사용하는 tex2D 형태의 텍스쳐 좌표에서도 계산은 동일하다. 각 레벨에서 계산되는 $P_i^{(s)}$ 의 개수(PC_s)는 $PC_s = N'/2^s - 1$ 이고 $P_i^{(s)}$ 의 저장이 시작되는 위치(PS_s)는 $s-1$ 단계까지의 PC_s 의 합 이다. $P_i^{(s)}$ 는 연속적으로 저장됨으로 저장이 시작되는 위치와 개수만 필요하다. 대응되는 텍스쳐 좌표의 범위(TW_s)는 $TW_s = N'/2^{s-1} - 1$ 이고 텍스쳐 좌표의 시작 위치(TS_s)는 $s-2$ 단계까지의 PC_s 의 합이다. 정리하면 다음과 같다.

$PC_s = N'/2^s - 1$,

$PS_s = PC_1 + .. + PC_{s-1} \quad (s \ge 2)$,

$PS_1 = 0$

$TW_s = N'/2^{s-1} - 1$,

$TS_s = PC_1 + .. + PC_{s-2} \quad (s \ge 3)$,

$TS_1 = 0$, $TS_2 = 0$

glTexCoord2f(TS_s, TS_s);

glVertex2f(PS_s, PS_s);

glTexCoord2f(TS_s, $TS_s + TW_s$);

glVertex2f(PS_s, $PS_s + PC_s$);

위의 OpenGL 코드와 같이 텍스쳐 좌표를 매핑시킴으로써 정점 프로세서(vertex processor)나 프래그먼트 프로세서(fragment processor)에서 프로그래밍으로 계산하지 않고도 인덱스 계산이 가능하다. 따라서 두 프로세서에 작업 부하가 생기지 않는다. 이는 인덱스들이 2의 배수로써 일정한 증가 값을 가지기 때문에 보간(interpolation) 계산에 의한 텍스쳐 좌표 매핑만으로 인덱스 계산이 충분하기 때문이다. 해 x의 계산에서 $P_i^{(s)}$ 데이타를 읽어오기 위한 인덱스 계산도 이와 유사하다. 이러한 인덱스 계산은 N에 따라 정해지는 상수 값들 이므로 GPU 구현의 초기화 단계에서 미리 계산하여 사용하면 편리하다.

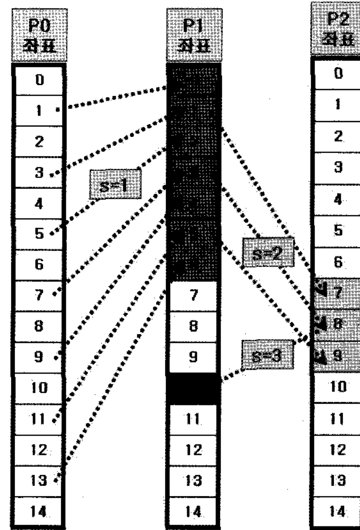


그림 4 텍스쳐 좌표 매핑

지금까지는 1차원 미분 방정식에서 유도된 삼중대각행렬 시스템 풀이에 대해서 설명하였다. 2차원과 3차원 미분 방정식에서 유도되는 삼중대각행렬 시스템은 다음 그림 5와 그림 6과 같이 1차원 풀이를 N번 또는 N×N 번 반복함으로써 풀 수 있다. 이는 그림 6에서 보여 지듯이 삼중대각행렬 시스템의 풀이는 y-축(j), z-축(h)에 대해서 독립적이기 때문이다. 그림 7과 그림 8은 2차원 미분 방정식의 삼중대각행렬 시스템에 대한 GPU 구현

$j = 1,..,N$

for s ← 1 to q-1
for i ← 2^s to $N' - 2^s$, step = 2^s
$P_{i,j}^{(s)} = (a_{i,j}^{(s)}, b_{i,j}^{(s)}, c_{i,j}^{(s)}, k_{i,j}^{(s)})$ compute

for s ← q to 1, step=-1
for i ← 2^{s-1} to $N' - 2^{s-1}$, step = 2^{s-1}
$x_{i,j}$ compute

} 1차원 풀이 N번 반복

그림 5 2차 미분 방정식의 삼중대각행렬 시스템의 풀이

$h = 1,..,N$
$j = 1,..,N$

for s ← 1 to q-1
for i ← 2^s to $N' - 2^s$, step = 2^s
$P_{i,j,h}^{(s)} = (a_{i,j,h}^{(s)}, b_{i,j,h}^{(s)}, c_{i,j,h}^{(s)}, k_{i,j,h}^{(s)})$ compute

for s ← q to 1, step=-1
for i ← 2^{s-1} to $N' - 2^{s-1}$, step = 2^{s-1}
$x_{i,j,h}$ compute

} 1차원 풀이 N×N번 반복

그림 6 3차 미분 방정식의 삼중대각행렬 시스템의 풀이

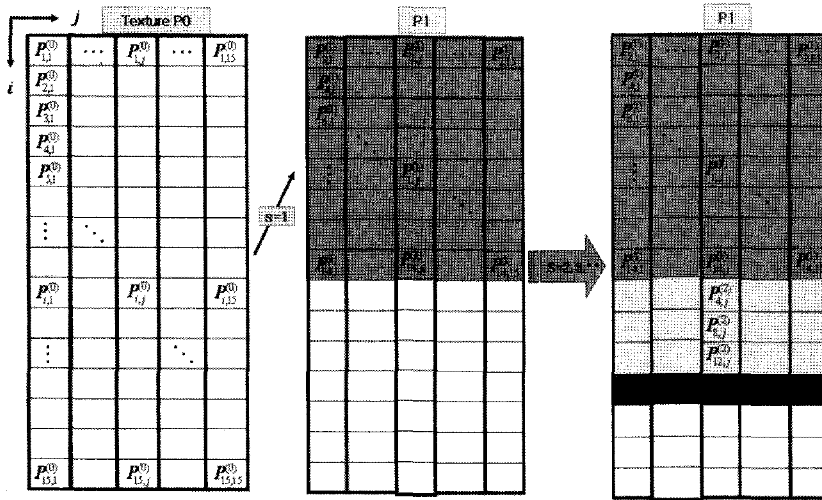


그림 7 2차원 미분 방정식의 삼중대각행렬 시스템 풀이 : P 벡터 계산

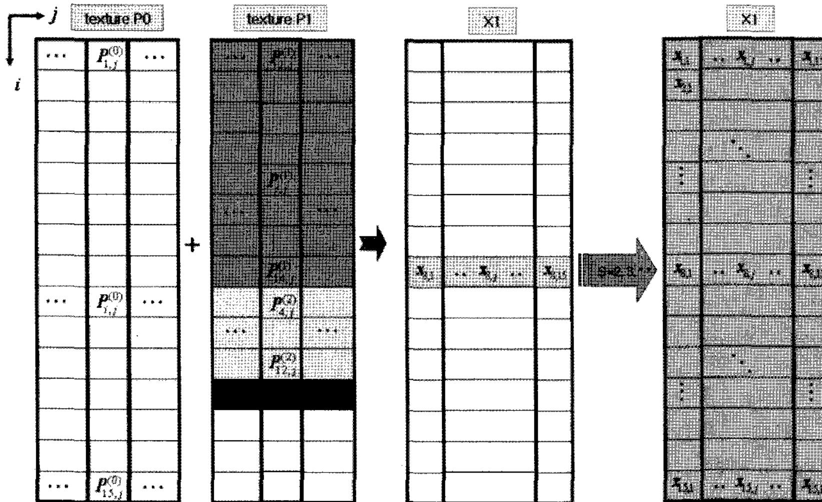


그림 8 2차원 미분 방정식의 삼중대각행렬 시스템 풀이 : 해 × 계산

과정을 보여준다. 동일한 계산을 하는 N개의 열(column)을 붙여 놓은 것과 같으므로 2차원 텍스쳐와 2차원 버퍼를 사용하는 것을 제외하고는 1차원의 계산 과정과 동일하다. 즉, i-행에 있는 N개의 모든 요소들이 동일한 방법으로 계산된다. 3차원의 경우는 입방체(cube) 맵 텍스쳐에 데이타를 저장하여 계산할 수도 있지만 2차원 텍스쳐를 이용하는 것이 성능이 좋으므로 그림 9와 같이 2차원 텍스쳐에 데이타를 저장하여 계산할 수 있다. 본 논문에서는 1차원 미분 방정식의 구현을 그림 2와 그림 3에서 보여 지듯이 1차원 텍스쳐를 사용하여 설명하였다. 2차원 텍스쳐를 사용하기 위해서는 인덱스 계산 과정이 필요하지만 본 논문에서는 생략하였다. 대부분의 자연 현상은 2차원이나 3차원의 미분 방정식으로 표현되며 GPU에서의 물리 기반 시뮬레이션 또한 2차원이나 3차원의 시뮬레이션에 관심이 있다.

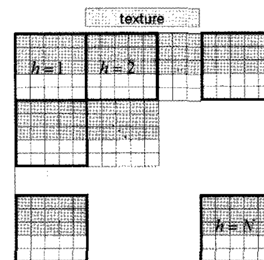


그림 9 3차원 미분 방정식의 삼중대각행렬 시스템 풀이를 위한 텍스쳐 구조

4. 실험 결과 및 분석

행렬 곱 계산이나 미분 방정식 풀이와 같은 계산은 텍스쳐 패치(texture fetch) 성능이 전체 계산에 큰 영향을 미친다[15]. 계산 시간이 텍스쳐 패치량과 정비례하는 것은 아니지만 이를 근거로 계산 성능을 예측할 수 있다. 따라서 본 논문에서는 GPU 구현에 있어 conjugate gradient(CG) 방법과 cyclic reduction(CR) 방법의 텍스쳐 패치량을 비교하였다. 표 1은 계산 과정을 보여준다. 식 (17)의 수식이 그림 10과 같은 Cg 코드로 표현된다고 가정할 때, 패치량 계산은 하나의 실수 데이타 요소를 가지는 스칼라(scalar) 텍스쳐를 사용하는 경우에 계산 영역 N에 대해서 x, p 두개의 텍스쳐 패치가 발생한다고 보았다. 그리고 packing 기법을 사용하여 실수 데이타 4개의 요소를 가지는 벡터(vector) 텍스쳐를 사용하는 경우라면 계산 영역 N×1/4에 대해서 2개의 패치가 발생하여 N영역에 대해서는 1/2개의 패치가 발생하는 것으로 계산하여 비교하였다. 표 1에서 보여지듯이 cyclic reduction 방법의 텍스쳐 패치량은 conjugate gradient 방법의 반복횟수가 1회인 경우보다 적음을 알 수 있다.

그림 11은 삼중대각행렬 시스템의 풀이 방법들에 대한 계산 시간을 보여준다. CPU에서의 계산은 Thomas 방법, GPU에서의 계산은 conjugate gradient 방법과 cyclic reduction 방법에 대해서 비교하였다. 이는 Pentium Ⅳ 3.0GHz, 1024MB 주메모리, PCIE 버스, 256MB 그래픽 메모리를 장착한 NVIDIA GeForce 6800의 GPU에서 실험되었으며 OpenGL과 Cg 셰이딩 언어로 구현하였다. GPU에서 cyclic reduction 방법은 conjugate gradient 방법의 반복 횟수가 2회인 경우보다 빠름을 볼 수 있다. 또한 conjugate gradient 방법으로 삼중대각행렬 시스템의 풀이를 하는 경우 N이 작을 때(그림 11(a)), GPU 구현에 의한 성능 향상을 기대하기 어려움을 볼 수 있다.

```
void cg_program( float2  coord  : TEX0,
                 out float   NewX  : COLOR,
                 uniform float  alpha,
                 uniform samplerRECT  X,
                 uniform samplerRECT  P )
{
   float x =  f1texRECT(X, coord);
   float p =  f1texRECT(P, coord);
    NewX = x + alpha*p ;
}
```

그림 10 GPU 구현에서 식 (17)의 Cg 코드

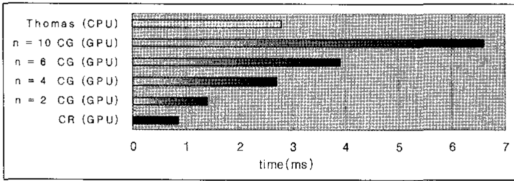
$$x_{i,j}^{n+1} = x_{i,j}^{n} + \alpha p_{i,j}, \qquad i,j \leftarrow 1,2,\ldots,N \tag{17}$$

다음은 2장에서 설명된 미분 방정식에 의한 물리 기반 시뮬레이션의 GPU 구현을 보여준다.

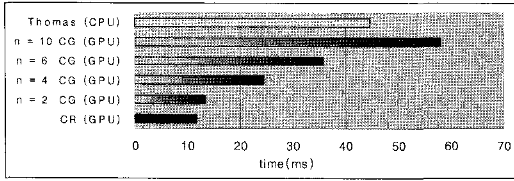
표 1 GPU구현에서 conjugate gradient와 cyclic reduction 방법의 텍스쳐 패치량 비교

Conjugate Gradient (CG) 알고리즘	계산 영역	텍스쳐 패치 갯수	Cyclic Reduction (CR) 알고리즘	계산 영역	텍스쳐 패치 갯수
$r^0 = k - Mx^0$	N	7	*for* $s \leftarrow 1:q-1$	$\sum_{s=1}^{q-1} 2^{q-s} = N - q \approx N$	12
for it $\leftarrow 1:n$			$p_{i,j}^{s}$ *compute*		
$\rho^{it-1} = (r^{it-1})^T \cdot r^{it-1}$	N	1	*for* i,j $\leftarrow 1:N$	N	6
	N×1/3	4	$x_{i,j}$ *compute*		
$\beta = \rho^{it-1} / \rho^{it-2}$					
$p^{it} = r^{it-1} + \beta p^{it-1}$	N	2			
$q^{it} = Mp^{it}$	N	6			
$\alpha = \rho^{it-1} / (p^{it-1})^T \cdot q^{it-1}$	N	1			
	N×1/3	4			
$x^{it} = x^{it-1} + \alpha p^{it}$	N	2			
$r^{it} = r^{it-1} - \alpha q^{it}$	N	2			
합	스칼라 텍스쳐	7 + 15.5×n	합	스칼라 텍스쳐	18
	벡터 텍스쳐	1.75 + 3.9×n		벡터 텍스쳐	4.5

* 텍스쳐 패치 갯수 : 계산 영역내의 각 요소 계산에 필요한 텍스쳐 패치 갯수



(a) N=256×256



(b) N=1024×1024

그림 11 삼중대각행렬 시스템 풀이 방법의 계산 시간 비교

[시뮬레이션 1]

그림 12는 식 (1)의 비정상 상태의 열전도 시뮬레이션의 실험환경이다. 경계에는 노이만(Neumann) 경계조건을 적용하였고 중심부 우측과 좌측에 균일한 온도의 발열을 두었다. 그림 13은 비정상 상태에서 온도의 변화를 보여준다. 그림 14는 그림 11에서 비교한 세 가지 방법으로 구현한 열전도 시뮬레이션의 성능을 FPS (Frame Per Second)로 보여준다. 계산 성능을 정확하게 비교하기 위해서 화면에 가시화하는 시간은 제외하였다. 계산 성능 면에서 그림 11의 삼중대각행렬 시스템 풀이의 결과와 비슷하다.

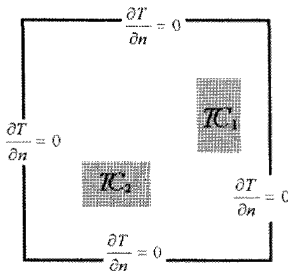
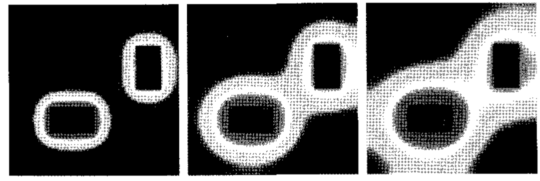
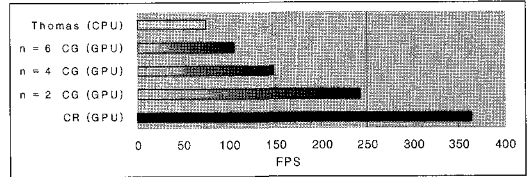


그림 12 열전도 시뮬레이션 환경

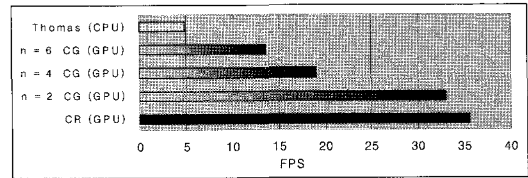


(a) (b) (c)

그림 13 열전도 시뮬레이션에서 시간에 따른 온도 변화



(a) N=256×256



(b) N= 1024×1024

그림 14 열전도 시뮬레이션의 성능 비교

[시뮬레이션 2]

그림 15~18은 식 (8)에 대해서 두 가지 상황을 시뮬레이션 하였다[16]. 첫 번째 실험 환경은 그림 15에서

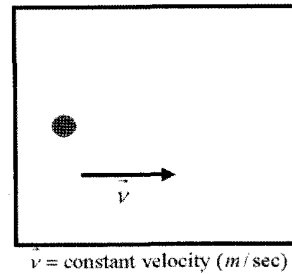
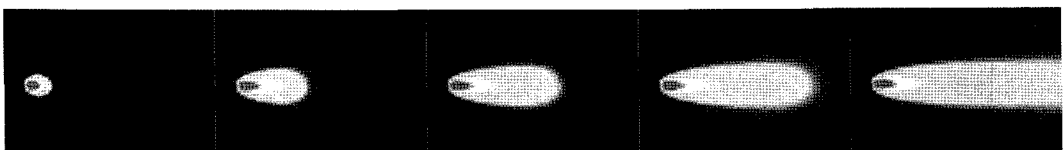


그림 15 오염 물질 확산 시뮬레이션 환경 : 오염 물질 연속 유입



(a) (b) (c) (d) (e)

그림 16 시간에 따른 오염 물질의 농도 변화 : 오염 물질 연속 유입

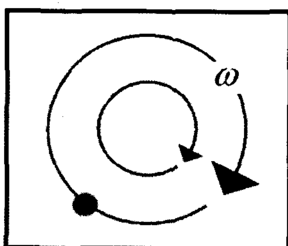


그림 17 오염 물질 확산 시뮬레이션 환경 : 오염 물질 짧은 기간 유입

보여주는 것처럼 y-축 방향의 속도는 0이고 x-축 방향으로만 일정한 속력이 주어진 환경에서 중앙 왼쪽 지점에 연속으로 오염 물질이 유입된다. 두 번째 실험 환경에서는 그림 17처럼 일정한 각속도를 가지는 속도의 환경에서 좌측 하단 지점에서 짧은 순간에 오염 물질이 유입되었다. 그림 16과 그림 18은 오염 물질의 농도가 시간이 경과함에 따라 변화하는 과정을 보여준다.

[시뮬레이션 3]

컴퓨터그래픽스 분야에서 쌍곡선형 미분 방정식에 ADI 방법을 적용한 경우로, 수심이 비교적 얕아서 물의 운동이 바닥의 영향을 받는 얕은 물(shallow water) 시뮬레이션이다[17]. 얕은 물 방정식(shallow water equation)에 대한 근사해를 구하는 것으로 파동 속도(wave velocity)가 $\sqrt{gd}$ 인 파동 방정식과 유사하며 수식은 다음과 같다.

$$\frac{\partial^2 h}{\partial t^2} = gd\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right) \tag{18}$$

여기서 g는 중력가속도 이고 d는 수심(water depth)으로 다음 수식과 같이 바닥의 지형에 따라 변화하는 값이다.

$d(x,y) = h(x,y) - b(x,y)$, b 는 바닥의 높이

식 (18)에 ADI 방법을 적용하면 다음과 같다.

• 단계 1 : $\frac{\partial^2 h}{\partial t^2} = gd\frac{\partial^2 h}{\partial x^2}$

• 단계 2 : $\frac{\partial^2 h}{\partial t^2} = gd\frac{\partial^2 h}{\partial y^2}$

단계 1의 경우 j-격자 선에 대해서 다음과 같은 삼중 대각행렬을 얻는다.

$$\begin{bmatrix} e_0 & f_0 & & & & & \\ f_0 & e_1 & f_1 & & & & \\ & f_1 & e_2 & & & & \\ & & & \bullet & & & \\ & & & & \bullet & & \\ & & & & e_{n-3} & f_{n-3} & \\ & & & & f_{n-3} & e_{n-2} & f_{n-2} \\ & & & & & f_{n-2} & e_{n-1} \end{bmatrix}$$

$$e_i = 1 + g(\Delta t)^2\left(\frac{d_{i-1,j} + d_{i,j} + d_{i+1,j}}{2(\Delta h)^2}\right),$$

$$f_i = -g(\Delta t)^2\left(\frac{d_{i,j} + d_{i+1,j}}{2(\Delta h)^2}\right)$$

이전 시뮬레이션에서는 행렬의 계수가 변하지 않기 때문에 시뮬레이션 초기 단계에서 미리 계산하여 시뮬레이션이 진행됨에 따라 동일한 행렬을 사용하였다. 얕은 물 시뮬레이션에서는 시뮬레이션의 매 시간마다 수심이 변화하기 때문에 매 시간마다 행렬 계수의 계산이 필요하다. 그림 19는 잔잔히 흐르는 얇은 물에 대각 방향으로 외부에서 힘이 가해지는 얕은 물 시뮬레이션의 결과 영상이다.

[성능 분석]

시뮬레이션 1과 시뮬레이션 2의 경우에 N=1024×1024의 계산 영역에서 cyclic reduction 방법을 이용할 때 가시화를 포함하여 35 FPS의 성능을 보여주었다. 시뮬레이션 3에서는 매 시간마다 행렬 계수의 계산과 3차원 가시화를 위한 과정이 추가적으로 필요하다. 물 표면의 높이와 물 표면의 높이로 부터 계산된 법선 벡터(normal vector) 데이타에 대해서 VBO (vertex buffer object)와 PBO(pixel buffer object)를 사용하여 랜더링하는 과정이 필요하다[18]. 따라서 N=1024×1024의 계산 영역에 대해서 17 FPS의 성능을 보여주었다.

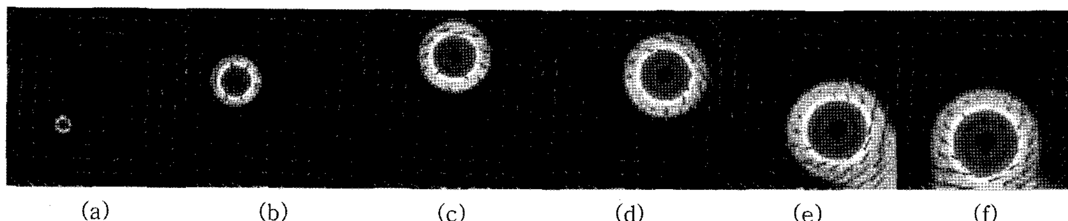


그림 18 시간에 따른 오염 물질의 농도 변화 : 오염 물질 짧은 기간 유입

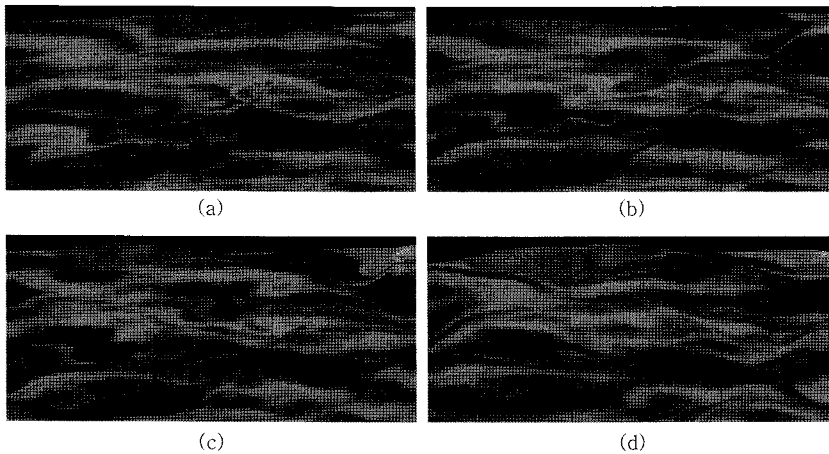


그림 19 얕은 물 시뮬레이션 결과 영상

5. 결론 및 향후 연구 과제

본 논문에서는 그래픽 하드웨어의 급속한 발전으로 인하여 범용 계산의 자원으로 급부상하고 있는 그래픽 프로세서 유닛(GPU)에서, 삼중대각행렬 시스템 풀이를 빠르게 구현할 수 있는 방법을 제안하였다. 벡터 프로세서에서 삼중대각행렬 시스템 풀이 방법으로 널리 사용되는 cyclic reduction의 두 알고리즘 중에서 O(N) 시간 복잡도를 가지는 직렬 cyclic reduction 알고리즘을 GPU에서 구현함으로써 GPU에서 삼중대각행렬 시스템 풀이에 상당한 성능 향상을 얻을 수 있었다. 또한 삼중대각행렬 시스템 풀이를 포함하는 물리 기반 시뮬레이션에 적용하여 계산영역이 1024×1024 격자인 시뮬레이션에서 초당 35프레임 이상의 놀라운 성능을 보여주었다.

GPU의 구조적 특성 때문에, CPU 구현에서는 좋은 성능을 보이나 GPU 구현에는 적합하지 않거나 GPU 구현으로 인한 성능 향상을 기대하기 어려운 알고리즘들이 있다. 반면에 벡터 프로세서(vector processor)나 다중 프로세서(multiprocessor)에서 사용되는 알고리즘처럼 특화된 프로세서에 적합한 알고리즘이기 때문에 일반적으로 널리 알려져 있지 않지만 GPU 구현에 적합한 알고리즘들이 많이 있을 것으로 예상한다. 앞으로도 우리는 다양한 분야에서 GPU 구현에 적합한 알고리즘들을 찾아서 GPU에서의 범용 계산에 활용하고자 한다.

참 고 문 헌

[1] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," In proceeding of the ACM SIGGRAPH/Eurographics conference on Graphics hardware, pp. 41-50, 2003.

[2] J. Kruger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," ACM Transactions on Graphics 22(3), pp. 908-916, 2003.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparce Matrix Solvers on the GPU: conjugate gradients and multigrid," ACM Transactions on Graphics 22(3), pp. 917-924, 2003.

[4] N. Goodnight, C. Woolley, G. Luebke, G. humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," In Graphics Hardware 2003, pp. 102-111.

[5] M. Harris, Fast fluid dynamics simulation on the GPUs, In GPU Gems, pp. 637-665, Addition Wesley, 2004.

[6] T. Kim, M. C. Lin, "Visual simulation of ice crystal growth," In 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 86-97.

[7] C. Zeller, "Cloth simulation on the GPU," In ACM SIGGRAPH Conference Abstracta and Applications, 2005.

[8] I. Viola, A. Kanitsar, M. E. Groller, "Hardware-based nonlinear filtering and segmentation using high-level shading languages," In IEEE Visualization 2003, pp. 309-316.

[9] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, "Fast computation of database operations using graphics processors," In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 215-226, 2004.

[10] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., Numerical recipes in C. The Art of scientific Computing. Cambridge University Press, 1992.

[11] R. W. Hockney and C. R. Jesshope, Parallel Computers, Adam and Hilger Ltd, Bristol, 1981,

Chapter 5.
[12] D. Heller, "A survey of Parallel Algorithms in Numerical Algebra," SIAM Review, vol. 20, pp. 740-777, 1978.
[13] S. Kotake and K. Hijikata, Numerical Simulation of Heat Transfer and Fluid Flow on a Personal Computer, Elsevier, 1993.
[14] Peaceman, D. W., and H. H. Rachford, J., "Numerical solution of parabolic and elliptic differential equations," J. Soc. Indust. Appl. Math., 3, pp. 28-41, 1955.
[15] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," Graphics Hardware, 2004.
[16] J. K. Lee, J. Y. Kim, and H. S. Kim, "Simulation of 2-D pollutant transport phenomena using modified characteristic method," Proc. of 29th IAHR Congress, 2001.
[17] M. Kass and G. Miller, "Rapid, Stable Fluid Dynamics for Computer Graphics," Computer Graphics, 24(4), pp.49-57, 1990.
[18] http://www.developer.nvidia.com

김 영 희
정보과학회논문지 : 시스템 및 이론
제 32 권 제 9 호 참조

이 성 기
정보과학회논문지 : 시스템 및 이론
제 32 권 제 9 호 참조