CIS 607 Final Project

Steven Walton
swalton2@cs.uoregon.edu

Herein contains my final project for CIS607. This paper contains all the code and notes related to the assignment (with section 3 being the "bonus" question as stated by the latest email). The code can also be found at `https://github.com/stevenwalton/CIS607-GPUProgramming` for added convenience.

1 Create *my_solvers.jl*

Create *my_solvers.jl* and include a function *conj_grad()* that performs the conjugate gradient (CG) algo- rithm (serial, CPU). *conj_grad()* should take in a positive definite matrix A, an initial guess, a right-hand-side vector b, a tolerance ϵ (set to 10^{-6}) and a maximum number of iterations $iter_{max}$ (normally set to a multiple of the length of b), and return an approximate solution $\tilde{x}$ to $Ax = b$, such that the relative error $err_R \leq \epsilon$ where

$$err_R = \frac{||A\tilde{x} - b||}{||\tilde{x}||}$$

where $||\cdot||$ is the 2-norm, i.e. $||x|| = \sqrt{x^T x}$. (Hint: one way to generate a positive definite $N \times N$ matrix is to define $A = I + B^T B$, as suggested above. A common choice for initial condition is the vector of all zeros).

1.1 My Code: *my_solvers.jl*

Note that the actual code contains a few comments and a block commented out section for testing this code

```
using LinearAlgebra
using CUDA

function conj_grad(A, x, b, epsilon, iter_max)
    r = b - A*x
    err = norm(r)
    if err < epsilon * norm(x)
        return x
    end
    p = r
    for k = 1:iter_max
        r_old = r
        alpha = (r' * r) / (p' * A * p)
        x .= x + alpha * p
        r = r - alpha * A * p
        err = norm(r)
        if err < epsilon * norm(x)
            return x
        end
        beta = (r' * r) / (r_old' * r_old)
        p = r + beta * p
    end
    return x
end

function norm_cuda!(err, x)
    for i = 1:length(x)
        err[1] = err[1] + (x[i] * x[i])
    end
    err = CUDA.sqrt(err[1])
    return nothing
end

function conj_grad_cuda!(A, x, b, epsilon, iter_max)
    r = b - A * x
    err = CuArray([0.])
    @cuda norm_cuda!(err, r)
    err2 = CuArray([0.])
    @cuda norm_cuda!(err2, x)
    if any(err .< epsilon .* err2)
        return
    end
    p = copy(r)
    for k = 1:iter_max
        r_old = copy(r)
        alpha = (r' * r) / (p' * A * p)
        x .= x + alpha * p
        @cuda norm_cuda!(err, r)
        @cuda norm_cuda!(err2, x)
        if any(err .< epsilon .* err2)
            return
        end
        beta = (r' * r) / (r_old' * r_old)
        p = r + beta * p
    end
    return
end
```

2 Poisson Equation

The 1-dimensional Poisson equation with Dirichlet boundary conditions is given by

$$\begin{aligned} -u_{xx} &= F(x), & 0 \le x \le 1 \\ u(0) &= 0 \\ u(1) &= 0 \end{aligned}$$

where the source $F(x) = \pi^2 \sin(\pi x)$

2.1 a) *poisson1D.jl*

Create a new Julia script and call it *poisson1D.jl* and include *my_solvers.jl*. To solve the 1D Poisson equation numerically, discretize in space with N total nodes using the standard (second-order accurate) centered difference approximation to u_{xx}, and arrive at a system of equations $Au = b$, where A is positive definite.

2.1.1 My Code: *poisson1D.jl*

Note here that we time the whole block to get better averaging, though each step size will take a different amount of time. But this will better compare with the GPU code

```julia
include("my_solvers.jl")

using LinearAlgebra
using SparseArrays
using Printf
using CUDA

function get_error(solution, input, step)
    u_total = [0; input; 0]
    return sqrt((u_total - solution)' * (u_total - solution)) * sqrt(step)
end

function poisson(h, eps)
    N = Integer(1/h + 1) # Total number of nodes
    m = N - 2 # total interior nodes
    x0 = zeros(m)
    x = 0:h:1
    exact = sin.(pi*x)
    A = zeros(m, m)
    for i = 1:m
        A[i, i] = 2/h^2
    end

    for i = 1:m-1
        A[i, i+1] = -1/h^2
        A[i+1, i] = -1/h^2
    end

    # make vector b
    b = Array{Float32}(undef, m)
    for i = 1:m
        b[i] =  pi^2 * sin(pi * x[i])
    end

    u_int = conj_grad(A, x0, b, eps, N^2)
    error = get_error(exact, u_int, h)

    # Half
    hd2 = h/2
    N_half = Integer(1 / hd2 + 1)
    m_half = N_half - 2
    x0_half = zeros(m_half)
    x_half = 0:hd2:1
    exact_half = sin.(pi * x_half)
    A_half = zeros(m_half, m_half)
    for i = 1:m_half
        A_half[i, i] = 2/hd2^2
    end
    for i = 1:m_half-1
        A_half[i, i+1] = -1/hd2^2
        A_half[i+1, i] = -1/hd2^2
    end

    # make vector b
    b_half = Array{Float32}(undef, m_half)
    for i = 1:m_half
        b_half[i] =  pi^2 * sin(pi * x_half[i])
    end
    u_int_half = conj_grad(A_half, x0_half, b_half, eps, N_half^2)
    error_half = get_error(exact_half, u_int_half, hd2)

    @printf "%f\t %f\t %f\t %f\n" h error (error/error_half) log2(error)
end

eps = 1e-9
h_list = [0.1, 0.05, 0.025, 0.0125]

@printf "dx\t\t error_dx\t ratio\t\t rate\n"
@time begin
    for i = 1:length(h_list)
        h = h_list[i]
        poisson(h, eps)
    end
end
```

2.2 b) Calling *conj_grad.jl*

Call *conj_grad()* to solve the system in part (a). Note that the exact solution is given by $u(x) = \sin(\pi x)$. Confirm that your numerical solution is converging in space at the correct rate (rate ≈ 2) by computing a convergence table. Where the error is the discrete L$_2$-norm of difference $u(x_j) - u_j$ (i.e. the exact solution $u(x)$ evaluated on the spatial grid and the numerical solution u_j). I will provide more details in class

2.2.1 Filling in the table

The total execution time took 1.443953 seconds (3.99 M allocations: 265.980 MiB

		2nd Order	
Δx	error$_{\Delta x}$	ratio = error$_{\Delta x}$/error$_{\Delta x/2}$	rate = log$_2$(ratio)
0.1	0.055232	2.232350	-4.178364
0.05	0.024741	2.064825	-5.336927
0.025	0.011982	2.016699	-6.382947
0.0125	0.005942	2.004207	-7.394943

Included in graph 1 is the convergence graphs for the CPU code. Note that we have inverted the x-axis for easier interpretation.

3 GPU Functionality (bonus!)

Add GPU functionality to the conjugate gradient method in part 1. Time the GPU code and compare to the serial version for several problems sizes (e.g. $N = 10^2, 10^3, \cdots$) Profile it using nvprof or Nsight. Comment on performance results.

3.1 My GPU Code

Note that I do not think my code is completely correct. This is because my code does not seem to be converging. As far as I can tell the functions are the same. I also check (see commented code) that the matrices are the same and that things were being allocated correctly. I am less sure how to debug the poisson code as I am unable to print out statements. I also found it frustrating that I was unable to allocate memory within a function, which is why this code is formatted differently than the CPU code. I'm not sure if there is a better way to do this but interested in knowing if there is.

Code to call the conjugate gradient (this is in *poisson1D.jl*)

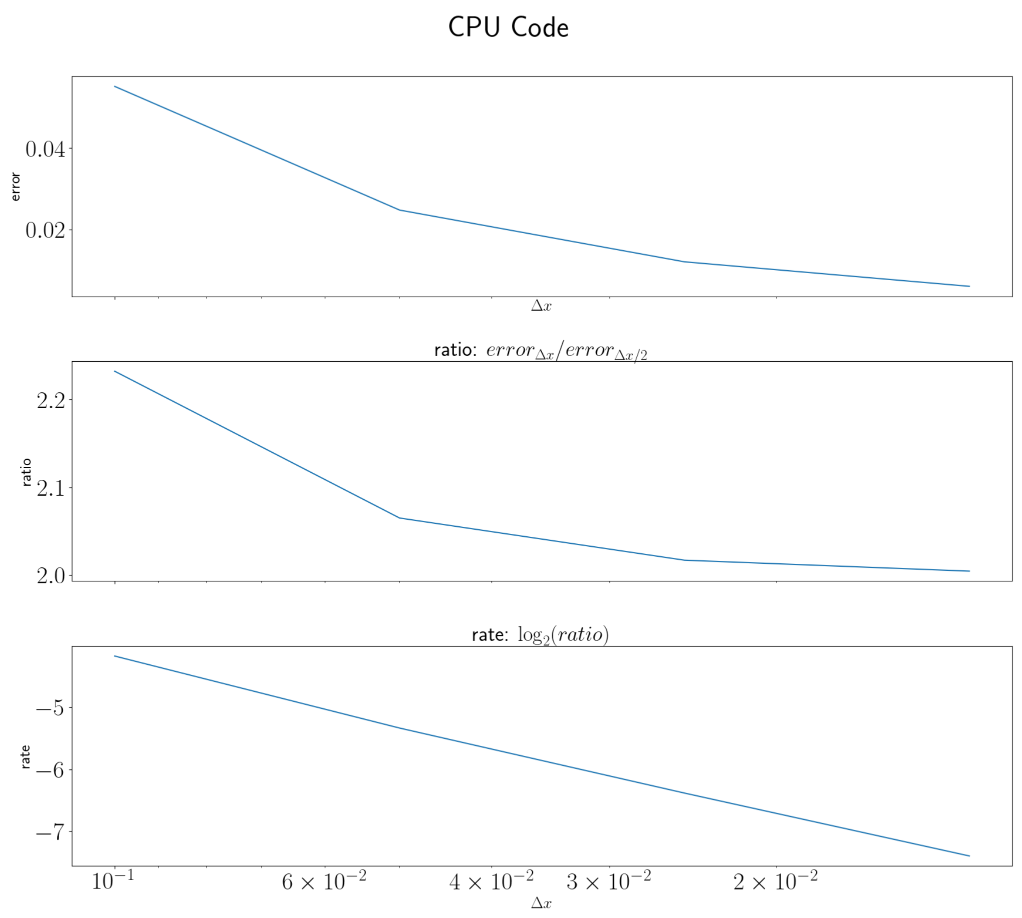

Figure 1: Graphs for CPU code

```
include("my_solvers.jl")

using LinearAlgebra
using SparseArrays
#using Plots
using Printf
using CUDA

function fillA!(A, h)
    m = size(A)[1]
    for i = 1:m
        A[i, i] = 2 / h^2
    end
    for i = 1:m-1
        A[i, i+1] = -1 / h^2
        A[i+1, i] = -1 / h^2
    end
    return nothing
end

function get_error_cuda!(error, solution, input, step)
    u_total = CuArray([0; input; 0])
    error = CUDA.sqrt((u_total - solution)' * (u_total - solution)) * sqrt(step)
```

```julia
    return nothing
end


eps = 1e-9
h_list = [0.1, 0.05, 0.025, 0.0125]

@time begin
    for i = 1:length(h_list)
        h = h_list[i]
        N = Integer(1 / h + 1)
        m = N-2
        x0 = CUDA.zeros(m)
        x_host = 0:h:1
        x = x_host |> cu
        exact = sin.(pi * x_host)
        A = CUDA.zeros(m, m)
        @cuda fillA!(A, h)
        b_host = zeros(m)
        for j = 1:m
            b_host[j] = pi^2 * sin(pi * x_host[j])
        end
        b = b_host |> cu
        synchronize()
        #A_host = zeros(m, m)
        #for i = 1:m
        #    A_host[i, i] = 2/h^2
        #end
        #for i = 1:m-1
        #    A_host[i, i+1] = -1/h^2
        #    A_host[i+1, i] = -1/h^2
        #end
        dummy = CUDA.zeros(m)
        #@cuda threads=128 blocks=32 knl_gemv!(dummy, A, x)
        #dummy2 = zeros(m)
        #gemv!(dummy2, A_host, x_host)
        #@show all((Array(dummy) - dummy2) .< 0.000000001)
        conj_grad_cuda!(A, x0, b, CuArray([eps]), N^2, dummy)
        error = get_error(exact, Array(x0), h)

        # half calculation
        h_half = h / 2
        N = Integer(1 / h_half + 1)
        m = N-2
        x0_half = CUDA.zeros(m)
        x_host = 0:h_half:1
        x = x_host |> cu
        exact = sin.(pi * (0:h_half:1))
        A = CUDA.zeros(m, m)
        @cuda fillA!(A, h_half)
        b_host = zeros(m)
        for j = 1:m
            b_host[j] = pi^2 * sin(pi * x_host[j])
        end
        b = b_host |> cu
        synchronize()
        dummy_half = CUDA.zeros(m)
        conj_grad_cuda!(A, x0_half, b, CuArray([eps]), N^2, dummy_half)
        error_half = get_error(exact, Array(x0_half), h_half)
        @printf "%f\t %f\t %f\t %f\n" h error (error/error_half) log2(error)
    end
end
```

The conjugate gradient GPU code (this is also in *my_solvers.jl*)

```
using LinearAlgebra
using CUDA

function norm_cuda!(err, x)
    for i = 1:length(x)
        err[1] = err[1] + (x[i] * x[i])
    end
    err = CUDA.sqrt(err[1])
    return nothing
end

function knl_gemv!(y, A, x)

    N = length(y)

    bid = blockIdx().x  # get the thread's block ID
    tid = threadIdx().x # get my thread ID
    dim = blockDim().x  # how many threads in each block

    i = dim * (bid - 1) + tid #unique global thread ID
        if i <= N
            for k = 1:N
                y[i] += A[i, k]*x[k]
            end
        end
    return nothing
end

function conj_grad_cuda!(A, x, b, epsilon, iter_max, dummy)
    threads_pb=32
    num_blocks = 5#cld(N, threads)
    @cuda threads=threads_pb blocks=num_blocks knl_gemv!(dummy, A, x)
    #r = b - A * x
    r = b - dummy
    err = CuArray([0.])
    @cuda norm_cuda!(err, r)
    err2 = CuArray([0.])
    @cuda norm_cuda!(err2, x)
    # Need .< Because this is an "array"
    if any(err .< epsilon .* err2)
        return
    end
    p = copy(r)
    for k = 1:iter_max
        r_old = copy(r)
        # Note that this appears to give the same results as when using
        # knl_gemv!()
        #alpha = (r' * r) / (p' * A * p)
        @cuda threads=threads_pb blocks=num_blocks knl_gemv!(dummy, A, p)
        alpha = (r' * r) / (p' * dummy)
        x .= x + alpha * p
        @cuda norm_cuda!(err, r)
        @cuda norm_cuda!(err2, x)
        # Need .< Because this is an "array"
        if any(err .< epsilon .* err2)
            return
        end
        beta = (r' * r) / (r_old' * r_old)
        p = r + beta * p
    end
    return
end
```

This code took 75.118018 seconds to run (65.55 M allocations: 3.899 GiB) I think the time difference is because the arrays were not sparse (evidence given by the 4GB memory allocation). I am unsure if CUDA can do this and the only way I know how to solve this is by writing a specific kernel for tridiagonal matrices and then converting A to a different format.

2nd Order			
Δx	error$_{\Delta x}$	ratio = error$_{\Delta x}$/error$_{\Delta x/2}$	rate = $\log_2$(ratio)
0.1	0.390671	0.793954	-1.355974
0.05	0.492058	0.843534	-1.023101
0.025	0.583328	0.910197	-0.777620
0.0125	0.640881	0.952675	-0.641871

The graph 2 shows that the code may be diverging rather than converging. This is unfortunately not what we want.

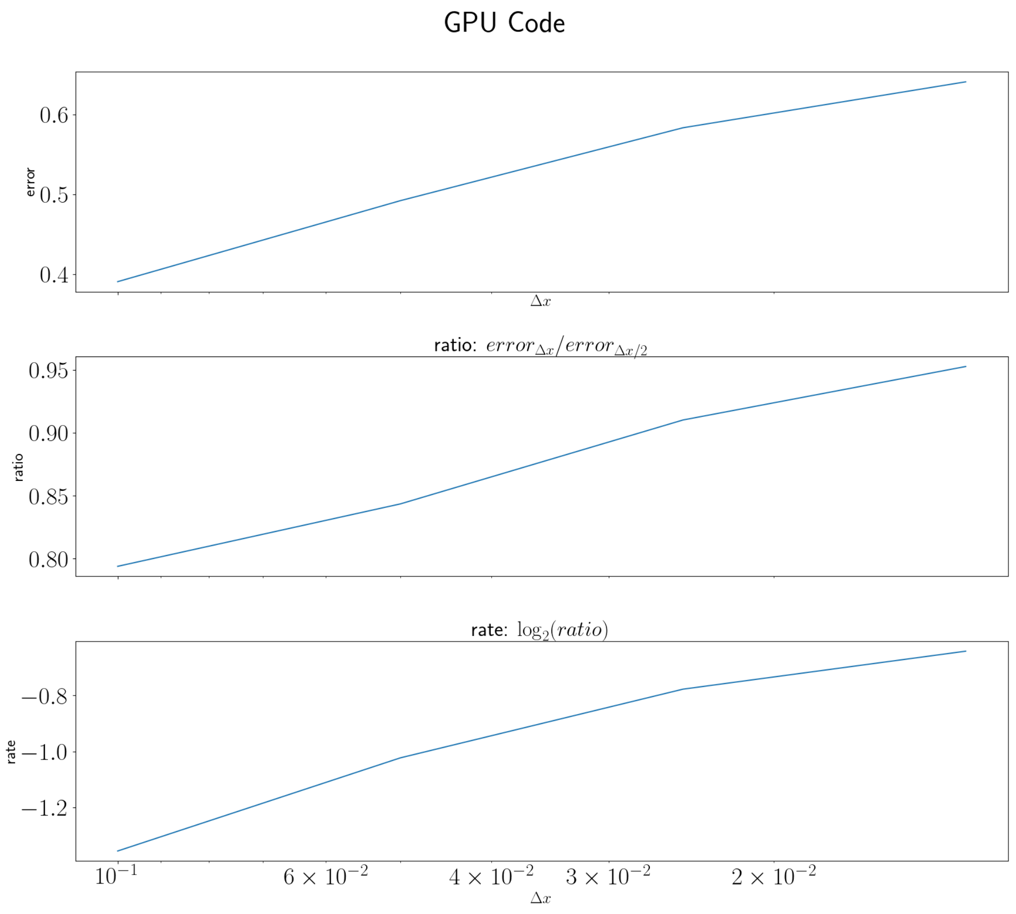

Figure 2: Graphs for GPU code