Displayed in Fig. 1 is the resulting convergence data recorded under successive refinements of the grid.

h	$\|u - u_h\|$	$\log_2\left(e_h/e_{\frac{h}{2}}\right)$
0.10000	0.000780	$\emptyset$
0.05000	0.000194	2.00698
0.02500	0.000048	2.00174
0.01250	0.000012	2.00044

Fig. 1: Convergence of approximate solution u_h computed on a GPU under successive mesh refinements.

Then, the GPU code is tested against the original serial code. We record the time it takes to complete all time stepping. Fig. 2 shows the result of the run time tests and we can see that the GPU code, though it slows down on larger inputs, performs better than the CPU code on large inputs.

h	CPU run time (s)	GPU run time (s)
0.005	0.004179153	0.003549657
0.001	0.119557278	0.016512015
0.0002	2.922218388	0.087218712s

Fig. 2: Run time comparison for GPU and CPU codes.

my_forward _EulerGPU!()

```julia
using SparseArrays
using CuArrays
using CUDAnative
using CUDAdrv: synchronize
include("addVectors.jl")
include("rates.jl")


# compute y = Ax + b
function knl_gemv_naive!(A, x, b)
    N = size(A, 1)

    @assert length(x) == N
    @assert length(b) == N

    y = zeros(N)

    for i = 1:N
        for j = 1:N
            y[i] += A[i, j] * x[j]# + b[i]/N
        end
        # which is faster, including bᵢ in the j loop or appending it
        # just outside the j loop by
        y[i] += b[i]
    end

    return y


end

function knl_gemv!(b, A, x)
    N = size(A,1)

    @assert length(x) == N
    @assert length(b) == N

    bidx = blockIdx().x  # get the thread's block ID
    tidx = threadIdx().x # get my thread ID
    dimx = blockDim().x  # how many threads in each block

    bidy = blockIdx().y  # get the thread's block ID
    tidy = threadIdx().y # get my thread ID
```

```
    dimy = blockDim().y  # how many threads in each block


    # figure out what work i need to do
    i = dimx * (bidx - 1) + tidx
    j = dimy * (bidy - 1) + tidy

    if i <= size(A, 1) && j <= size(x, 2)
        for k = 1:N
            b[i, j] += A[i, k] * x[k, j]
        end
    end


end

#=
let

    M = 10
    N = M
    Q = 1

    A = rand(M, N)
    x = rand(N, Q)
    b = rand(N, Q)

    d_A = CuArray(A)
    d_x = CuArray(x)
    d_b = CuArray(b)
    d_bcopy = similar(d_b)



    num_threads_per_block_x = 32
    num_threads_per_block_y = 32
    thd_tuple = (num_threads_per_block_x, num_threads_per_block_y)

    num_blocks_x = cld(M, num_threads_per_block_x)
    num_blocks_y = cld(Q, num_threads_per_block_y)

    #matmul!(C, A, B)
    #fake_knl_matmul!(C, A, B, num_threads_per_block_x, num_threads_per_block_y, num_blocks_

    @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_bcopy, d_A,
```

```
    synchronize()

    t_device = @elapsed begin
        @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A,
        synchronize()
    end
    @show t_device

    t_host = @elapsed begin
        C0 = A * x + b
    end


    @show t_host


    #@show norm(CuArray(C0) - d_C)
    @assert CuArray(C0) ≈ d_b
end
=#

function F(x,t)
    return 2*(pi^2 -1)*exp(-2t)*sin.(pi*x)
end

function f(x)
    return sin.(pi*x)
end

function exact(x,t)
    return exp(-2t)*sin.(pi*x)
    #return exp(-pi^2 * t)*sin.(pi*x)
end

function time_dependent_heat_GPU(k, Δx, Δt, T, my_source, my_initial)
    N  = Integer(ceil((1-0)/Δx)) # N+1 total nodes, N-1 interior nodes
    x = 0:Δx:1
    t = 0:Δt:T
    M = Integer(ceil((T-0)/Δt)) # M+1 total temporal nodes

    λ = Δt/Δx^2

    # A is N+1 by N+1
    A = (1-2*λ*k)*sparse(1:N+1,1:N+1,ones(N+1), N+1, N+1) +
            (λ*k)*sparse(2:N+1,1:N,ones(N),N+1,N+1) +
```

```
            (λ*k)*sparse(1:N,2:N+1,ones(N),N+1,N+1)

    # initial conditions
    u = zeros(N+1,1)
    u .= my_initial.(x)
    u_serial = copy(u)

    d_A = CuArray(A)
    d_u = CuArray(u)
    d_b = CuArray(zeros(Float64, N+1))

    num_threads_per_block_x = 32
    num_threads_per_block_y = 32
    thd_tuple = (num_threads_per_block_x, num_threads_per_block_y)

    num_blocks_x = cld(size(A,1), num_threads_per_block_x)
    num_blocks_y = cld(size(u,2), num_threads_per_block_y)

    F(x, t[1])
    @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A, d_u)

    t_dev = @elapsed begin
    for n = 1:length(t)
        b = Δt * F(x, t[n])
        d_b = CuArray(b)
        @cuda threads = thd_tuple blocks = (num_blocks_x, num_blocks_y) knl_gemv!(d_b, d_A,
        synchronize()

        d_u .= d_b

        d_u[1,1] = 0
        d_u[length(x),1] = 0
    end
    end
    @show t_dev

    t_serial = @elapsed begin
    for n = 1:length(t)
        b_serial = Δt * F(x, t[n])
        u_serial .= A*u_serial + b_serial

        u_serial[1] = 0
        u_serial[length(x)] = 0
    end
    end
```

```
    @show t_serial


    #Δx = 0.00125?



    return d_u
end

function my_forward_Euler!(Y, Δt, t, A, F, y, x, N, M)
    #M = length(t)
    #t = t1:Δt:tf
    #M = Integer(ceil((tf-t1)/Δt))
    Y[2:N,1] = y[:]

    for n = 1:M
        b = Δt * F(x[2:N],t[n])
        y[:] = A * y[:] + b
        Y[2:N,n] .= y[:]

    end

    #return (t, Y)

end


function (Δx)
    k = 2
    Δx = 0.000125


    λ = 0.1
   Δt = λ*Δx^2 / k
    T = 3*Δt

    x = 0:Δx:1
    t = 0:Δt:T

    uₕ = time_dependent_heat_GPU(k, Δx, Δt, T, F, f)

    uexact =CuArray( exact(x, t[length(t)]))
    #uexact = exact(x, t[length(t)])
    error =sqrt.( Δx * (transpose(uₕ-uexact) * (uₕ-uexact)))
```

```
    return error[1,1]
end

#convergence_rates(, 0.1, 4)
```